PARASOFT®

insure++®

User’s Guide

Version 6.1
UNIX

Parasoft Corporation
2031 S. Myrtle Ave.
Monrovia, CA 91016
Phone: (888) 305-0041
Fax: (626) 305-9048
E-mail: info@parasoft.com
URL: www.parasoft.com

PARASOFT END USER LICENSE AGREEMENT

REDISTRIBUTION NOT PERMITTED

This Agreement has 3 parts. Part I applies if you have not purchased a license to the accompanying software (the "SOFTWARE"). Part II applies if you have purchased a license to the SOFTWARE. Part III applies to all license grants. If you initially acquired a copy of the SOFTWARE without purchasing a license and you wish to purchase a license, contact Parasoft Corporation ("PARASOFT"):

(626) 305-0041

(888) 305-0041 (USA only)

(626) 305-9048 (Fax)

info@parasoft.com

http://www.parasoft.com

PART I -- TERMS APPLICABLE WHEN LICENSE FEES NOT (YET) PAID GRANT.

DISCLAIMER OF WARRANTY.

Free of charge SOFTWARE is provided on an "AS IS" basis, without warranty of any kind, including without limitation the warranties of merchantability, fitness for a particular purpose and non-infringement. The entire risk as to the quality and performance of the SOFTWARE is borne by you. Should the SOFTWARE prove defective, you and not PARASOFT assume the entire cost of any service and repair. This disclaimer of warranty constitutes an essential part of the agreement. SOME JURISDICTIONS DO NOT ALLOW EXCLUSIONS OF AN IMPLIED WARRANTY, SO THIS DISCLAIMER MAY NOT APPLY TO YOU AND YOU MAY HAVE OTHER LEGAL RIGHTS THAT VARY BY JURISDICTION.

PART II -- TERMS APPLICABLE WHEN LICENSE FEES PAID

GRANT OF LICENSE.

PARASOFT hereby grants you, and you accept, a limited license to use the enclosed electronic media, user manuals, and any related materials (collectively called the SOFTWARE in this AGREEMENT). You may install the SOFTWARE in only one location on a single disk or in one location on the temporary or permanent replacement of this disk. If you wish to install the SOFTWARE in multiple locations, you must either license an additional copy of the SOFTWARE from PARASOFT or request a multi-user license from PARASOFT. You may not transfer or sub-license, either temporarily or permanently, your right to use the SOFTWARE under this AGREEMENT without the prior written consent of PARASOFT.

LIMITED WARRANTY.

PARASOFT warrants for a period of thirty (30) days from the date of purchase, that under normal use, the material of the electronic media will not prove defective. If, during the thirty (30) day period, the software media shall prove defective, you may return them to PARASOFT for a replacement without charge.

THIS IS A LIMITED WARRANTY AND IT IS THE ONLY WARRANTY MADE BY PARASOFT. PARASOFT MAKES NO OTHER EXPRESS WARRANTY AND NO WARRANTY OF NONINFRINGEMENT OF THIRD PARTIES' RIGHTS. THE DURATION OF IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION, WARRANTIES OF MERCHANTABILITY AND OF FITNESS FOR A PARTICULAR PURPOSE, IS LIMITED TO THE ABOVE LIMITED WARRANTY PERIOD; SOME JURISDICTIONS DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO LIMITATIONS MAY NOT APPLY TO YOU. NO PARASOFT DEALER, AGENT, OR EMPLOYEE IS AUTHORIZED TO MAKE ANY MODIFICATIONS, EXTENSIONS, OR ADDITIONS TO THIS WARRANTY.

If any modifications are made to the SOFTWARE by you during the warranty period; if the media is subjected to accident, abuse, or improper use; or if you violate the terms of this Agreement, then this warranty shall immediately be terminated. This warranty shall not apply if the SOFTWARE is used on or in conjunction with hardware or software other than the unmodified version of hardware and software with which the SOFTWARE was designed to be used as described in the Documentation. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY HAVE OTHER LEGAL RIGHTS THAT VARY BY JURISDICTION.

YOUR ORIGINAL ELECTRONIC MEDIA/ARCHIVAL COPIES.

The electronic media enclosed contain an original PARASOFT label. Use the original electronic media to make "back-up" or "archival" copies for the purpose of running the SOFTWARE program. You should not use the original electronic media in your terminal except to create the archival copy. After recording the archival copies, place the original electronic media in a safe place. Other than these archival copies, you agree that no other copies of the SOFTWARE will be made.

TERM.

This AGREEMENT is effective from the day you install the SOFTWARE and continues until you return the original SOFTWARE to PARASOFT, in which case you must also certify in writing that you have destroyed any archival copies you may have recorded on any memory system or magnetic, electronic, or optical media and likewise any copies of the written materials.

CUSTOMER REGISTRATION.

PARASOFT may from time to time revise or update the SOFTWARE. These revisions will be made generally available at PARASOFT's discretion. Revisions or notification of revisions can only be provided to you if you have registered with a PARASOFT representative or on the Parasoft Web site. PARASOFT's customer services are available only to registered users.

PART III -- TERMS APPLICABLE TO ALL LICENSE GRANTS

SCOPE OF GRANT.

DERIVED PRODUCTS.

Products developed from the use of the SOFTWARE remain your property. No royalty fees or runtime licenses are required on said products.

PARASOFT'S RIGHTS.

You acknowledge that the SOFTWARE is the sole and exclusive property of PARASOFT. By accepting this agreement you do not become the owner of the SOFTWARE, but you do have the right to use the SOFTWARE in accordance with this AGREEMENT. You agree to use your best efforts and all reasonable steps to protect the SOFTWARE from use, reproduction, or distribution, except as authorized by this AGREEMENT. You agree not to disassemble, de-compile or otherwise reverse engineer the SOFTWARE.

SUITABILITY.

PARASOFT has worked hard to make this a quality product, however PARASOFT makes no warranties as to the suitability, accuracy, or operational characteristics of this SOFTWARE. The SOFTWARE is sold on an "as-is" basis.

EXCLUSIONS.

PARASOFT shall have no obligation to support SOFTWARE that is not the then current release.

TERMINATION OF AGREEMENT.

If any of the terms and conditions of this AGREEMENT are broken, this AGREEMENT will terminate automatically. Upon termination, you must return the software to PARASOFT or destroy all copies of the SOFTWARE and Documentation. At that time you must also certify, in writing, that you have not retained any copies of the SOFTWARE.

LIMITATION OF LIABILITY.

You agree that PARASOFT's liability for any damages to you or to any other party shall not exceed the license fee paid for the SOFTWARE.

PARASOFT WILL NOT BE RESPONSIBLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OF THE SOFTWARE ARISING OUT OF ANY BREACH OF THE WARRANTY, EVEN IF PARASOFT HAS BEEN ADVISED OF SUCH DAMAGES. THIS PRODUCT IS SOLD "AS-IS".

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

ENTIRE AGREEMENT.

This Agreement represents the complete agreement concerning this license and may be amended only by a writing executed by both parties. THE ACCEPTANCE OF ANY PURCHASE ORDER PLACED BY YOU IS EXPRESSLY MADE CONDITIONAL ON YOUR ASSENT TO THE TERMS SET FORTH HEREIN, AND NOT THOSE IN YOUR PURCHASE ORDER. If any provision of this Agreement is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This Agreement shall be governed by California law (except for conflict of law provisions).

All brand and product names are trademarks or registered trademarks of their respective holders.

Parasoft Corporation
2031 South Myrtle Avenue
Monrovia, CA 91016
Printed in the U.S.A, February 7, 2003

Table of Contents

Introduction

Getting Started

Using Insure++

Inuse

TCA

Reference

Error Codes

Index

Welcome!

C and C++ developers have a unique problem: many errors in their code don't manifest themselves during testing. Code with subtle problems such as memory corruption may run flawlessly on one machine, but crash on another. To find and fix such problems prior to release, you need a tool that works like an x-ray machine to expose the hidden defects in your code. You need Insure++®.

Insure++ saves you hours and hours of painstaking manual labor and wasted resources by automatically exposing such difficult-to-find problems as memory corruption, memory leaks, pointer errors, I/O errors, and more. With the click of a button or a simple command, Insure++ automatically uncovers the defects in your code - even those defects that were previously unknown.

Insure++'s Breakthrough Technologies

Insure++ detects more errors than any other tool because its technologies achieve the deepest possible understanding of the code under test and flush out even the most elusive problems.

Using patented Source Code Instrumentation (#5,581,696 and #6,085,029) and Runtime Pointer Tracking (#5,842,019) technologies, Insure++ develops a comprehensive knowledge of the software and all of its elements under test. During compilation, Insure++ inserts test and analysis functions around every line of source code. It builds a database of program elements, and then checks each data value and memory reference against the database at runtime to verify consistency and correctness.

Using these unique technologies, Insure++ thoroughly examines and tests your code from inside and out, including "rewriting" it through a process called "Mutation Testing," then reports errors and pinpoints their exact location. Insure++ also performs coverage analysis, clearly indicating which sections of the code were tested.

By integrating Insure++ into your development environment, you can save weeks of debugging time and prevent costly crashes from affecting your customers. You can also use Insure++ with other Parasoft tools to speed up debugging from the design phase all the way through testing and QA.

Pinpointing Programming Errors

Two of the most serious software-related problems are the time needed to debug (and therefore deliver) a product and the number of bugs that are not detected during testing and which are only found at customer sites.

These problems arise in many different ways. Insure++ finds a wide variety of programming and memory access errors, including:

- Memory corruption due to reading or writing beyond the valid areas of global, local, shared, and dynamically allocated objects.
- Operations on uninitialized, NULL, or "wild" pointers.
- Memory leaks.
- Errors allocating and freeing dynamic memory.
- String manipulation errors.
- Operations on pointers to unrelated data blocks.
- Invalid pointer operations.
- Incompatible variable declarations.
- Mismatched variable types in `printf` and `scanf` argument lists.

Insure++ does not use a "statistical" approach to trap memory reference errors. Instead, Insure++ checks each memory reference for validity when that reference is executed, including those to static (global) and stack, as well as dynamically allocated memory. When Insure++ finds a problem, it reports the name of related variables, the line of source code containing the error, a description of the error, and a stack trace.

Checking Calls to Libraries

Just as it does with memory reference errors, Insure++ finds the following library interface errors:

- Mismatched argument types or function declarations.
- Invalid parameters in library calls.
- Errors returned by library calls.

Insure++ understands standard UNIX system calls, the X Window system, Motif, and many other popular libraries. On each library call, Insure++ checks that every variable is of the correct type and is within its valid range.

If the source code for a third-party library is available, Insure++ can automatically check it if you rebuild the library with Insure++ as you do your own source code. If you don't have the source code, Insure++ includes utilities which allow you to make a "definition" of their interfaces. Once the interface is completely specified, these libraries will receive the same comprehensive checking that Insure++ provides for standard libraries.

Code Coverage Analysis with TCA

The Total Coverage Analysis (TCA) add-on works hand-in-hand with Insure++ to show you which parts of code you've tested and which you've missed. With TCA, you can stop wasting time testing the same parts of code over and over again and start exercising untested code instead.

Memory Optimization with Inuse

Your program may handle memory in real-time without any obvious problems, but only Inuse can tell you for sure. Find out where unseen leaks and other memory abuses are hurting your program with this graphical "memory visualization" tool.

Supported Platforms and Compilers

The platforms and compilers supported by Insure++ at the time this manual was printed are summarized in the table below:

		C++ Compilers					C Compilers	
Platform	**OS**	**cxx**	**xlc**	**aCC**	**CC**	**g++**	**gcc**	**cc**
AIX	5L		**X**			**X**	**X**	**X**
HP-UX (PA-RISC, IA-64)	11.x			**X**		**X**	**X**	**X**
Linux (x86, PPC, MIPS)	glibc 2.1+ for x86 and PPC, glibc 2.2+ for MIPS					**X**	**X**	
Solaris (SPARC)	7, 8, 9				**X**	**X**	**X**	**X**

The OS version listed above is the version under which Insure++ was built. Older OS versions may work, and newer versions will generally work.

If you have a different compiler, you may be able to customize Insure++ for your needs. Contact technical support (support@parasoft.com) for more details. Supported versions of the g++ compiler include 2.95.x, 3.0, 3.1, and 3.2.

New Features for Insure++ 6.1

Insure++ 6.1 for UNIX contains significant changes from previous versions that will help you debug faster and more efficiently. These include powerful new technologies and refined user interfaces.

New features/enhancements include:

- Improved runtime performance.
- Improved parser.
- Updated licensing scheme.
- General user enhancements.
- gcc/g++ 3.2 support.

Insure++ Installation, Startup, and Licensing

This section includes installation instructions for Insure++. If you are upgrading from an earlier version of Insure++, please consult the file *Release.notes* for brief descriptions of some major changes in this version. The amount of disk space required by Insure++ 6.1 depends on which system you are installing. The table below shows the approximate size and ARCH label for each supported platform. These labels will help you complete the installation.

System	ARCH	Disk Space (MB)
HP_UX 11.22 (HP IPF)	hp11_ia64	170
HP_UX 11.22 (HP IPF)	hp11	130
Linux (glibc 2.1 - glibc 2.2.5 x86)	linux2	90
Linux (glibc 2.2 - glibc 2.2.5)	linux_MIPS	120
Linux (glibc 2.1.3 - glibc 2.2.5)	linux_PPC	65
Solaris 7, 8, 9	solaris	180

Installing Insure++ involves the following steps:

Step 1. Create a Directory for the Insure++ Distribution

Step 2. Extract the CD-ROM Contents

Step 3. Extract the Installation Script

Step 4. Install Insure++

Step 5. Post-Installation Configuration

Step 6. Install a License

Step 7. Set the PARASOFT Environment Variable

Step 8. Modify Your PATH

Step 9. Modify Your Environment

Step 10. Running An Example

Each of these steps is fully explained below. Note: The following steps describe the process of installing Insure++ from a CD-ROM. Installation from the CD-ROM requires root privileges.

Step 1. Create a Directory for the Insure++ Distribution

Choose a location in which to install Insure++ and make this directory with a command such as:

```
mkdir <target directory>
```

For example: `mkdir /usr/local/parasoft`

Make sure you have write privileges to this <target directory>.

Important: It is recommended that you do not install Insure++ as root. When Insure++ is configured during installation, it must be configured for the development environment, so that the proper compilers will be configured. Be sure to use a directory where Insure++ can use the same compilers that you would normally use during development.

All subsequent steps must be performed in the new directory, so you should change to it now:

```
cd <target directory>
```

For example: `cd /usr/local/parasoft`

From now on we will assume that you have chosen to install the software in a directory called /usr/local/parasoft as indicated above. If you choose a different name, then modify the following commands appropriately.

Note for Linux Users: Chaperon records an absolute path during configuration. If you are using NFS or automounter, be sure that the mount paths on the NFS client are the same as on the NFS server.

Step 2. Extract the CD-ROM Contents

To mount the CD-ROM, you must be at the root:

```
su root
```

Create a /cdrom directory, if necessary:

```
mkdir /cdrom
```

The actual mount command is different on each platform we support. Please use the appropriate command for your system. Note that the following commands assume your CD-ROM drive is at SCSI ID 6. If your drive is at a different SCSI location, substitute the appropriate device file name for your CD-ROM drive.

System	Command
IBM AIX	mount -v cdrfs -r /dev/cd0 /cdrom
HP-UX	mount -F cdfs -r /dev/dsk/c0t6d0 /cdrom
Linux	mount -t iso9660 -o ro /dev/scd0 /cdrom
Solaris	mount -F hsfs -r /dev/dsk/c0t6d0s2 /cdrom

Change directory to the installation (target) directory you created in the first step.

```
cd <target directory>
```

For example: `cd /usr/local/parasoft`

Then copy the tar file for your platform to the current directory with the following command:

```
cp /cdrom/insure/tar/ins_$(ARCH).tar .
```

Unmount the CD-ROM with the following command:

```
umount /cdrom
```

You can now press the eject button on your CD-ROM drive to eject the CD before proceeding to Step 3.

Step 3. Extract the Installation Script

Make sure you have write privileges to the <target directory> before you start extracting installation files.

Extract the installation script with the command:

```
uncompress -c <tar_file> | tar xvf - install
```

for a compressed tar file, or

```
gzip -dc <tar_file> | tar xvf - install
```

for a gzipped tar file in which the name of the compressed or gzipped tar file supplied to you should be inserted in place of the text <tar_file>.

If you are installing from a CD-ROM and your tar file is not compressed, use the following command:

```
tar xvf <tar_file> install.
```

You are now ready to install Insure++ on your system using the provided installation script. Steps 4 through 11 will lead you through this procedure.

Step 4. Install Insure++

Insure++ includes an installation script that will help you install Insure++ on your system. To run the installation script, execute the command:

```
./install
```

The script first prints version and technical support information

```
Extracting installation scripts ...
Insure++ Installation Script Version 6.1 (06/04/02)
Copyright (C) 1997-2002 by Parasoft
Technical Support is available at:
E-mail: support@parasoft.com
Web:            http://www.parasoft.com
Telephone:(626) 305-0041
Fax:            (626) 305-9048
```

before asking you to confirm that you want to install Insure++ in the current directory. When the installation is completed, you will see the banner

```
* Installation of Insure++ 6.1 completed *
```

The Insure++ distribution consists of the following directories and files. ARCH below will be replaced in your distribution with your platform name, for example, sgi6.

Directory	**Contents**
`bin.ARCH`	Insure++ executables
`lib.ARCH`	Insure++ libraries and interfaces
`src.ARCH`	Insure++ interface source code
`examples`	Insure++ example programs and scripts
`insra`	Insra help files
`Inuse/`	Inuse help files
`man/`	Insure++ user's manual
`tca/`	TCA help files
`configure`	Insure++ compiler configuration script
`install`	Insure++ installation script
`FAQ.txt`	Insure++ Frequently Asked Questions

Note: The Insure++ example programs and scripts can be used for customizing your use of Insure++

Step 5. Post-Installation Configuration

The installation script will next lead you through a series of questions as it configures Insure++ for your system. The script allows you to determine:

- Which compilers will be used with Insure++. This step does not tell Insure++ which compiler to use when instrumenting and compiling your source code. You will still need to add an `insure++. compiler <compiler>` option to one of your `.psrc` files to specify which compiler should be used by Insure++. Therefore, you should answer "yes" to these questions for any compiler which you might use with Insure++ at some point in the future.
- Whether to send output to stderr or Insra (a GUI report viewer) by default. If you want to configure Insure++ for additional compilers later, you can execute the script `./configure` in the installation directory. Insure++ is now installed and configured on your system, but your system must be configured before use.

Step 6. Install a License

The procedure for installing a license depends on whether you are installing a machine-locked license or shared network license (using the Parasoft LicenseServer).

Machine-Locked Licenses

After installing and configuring the necessary files, the installation script will look for a valid license for Insure++. If one is not found, it will ask if you would like to install one. If so, the script will start pslic, the Parasoft License Manager.

If you get an error message from pslic saying that it cannot open a `.psrc` file, you should make sure that you have write permission in the

```
<target directory>
```

(`/usr/local/parasoft`) and/or run pslic as superuser. pslic will print out your machine and network id numbers. You should then phone, fax, or email this information to Parasoft. You will receive a license which you can enter using pslic. You can complete the remaining steps of the installation procedure without a license, but will not be able to use Insure++.

Once you have the license, run pslic:

Choose option "A" to add a license: `(A)dd a license`

The first item you will need to enter is the network or host ID number, which should be the same number printed by pslic. Next, you will be prompted to enter the expiration date, which you received from Parasoft. Finally, enter the password you were given.

To complete the license installation, select option "E" to exit and save changes: `(E)xit` and save changes

Shared Network Licenses

If you are using Parasoft's LicenseServer to manage a floating license, enter the following parameters in your `.psrc` file:

```
LicenseServer.host "hostname"
```

(Replace "hostname" with the name of the machine hosting LicenseServer; for example, `LicenseServer.host machine1`).

```
LicenseServer.port "port"
```

(Replace "port" with the port that LicenseServer is using; for example, `LicenseServer.port 2002`).

Insure++ can be run on a different host from the one registered with LicenseServer, as long as the `.psrc` file for the new machine is altered to include the necessary shared network license information.

For example: if the LicenseServer is running on HostA, and the user runs Insure++ from HostB, then add the following information in the `.psrc` file wherever Insure++ is installed in HostB:

```
LicenseServer.host HostA
LicenseServer.port 2002
registertool LicenseServer 1.0
```

Important: If you are using LicenseServer, you do not need to run pslic or enter anything in pslic.

Step 7. Set the PARASOFT Environment Variable

Note: This step is optional and is **NOT** recommended.

In most cases, you will not need to set this environment variable. However, you may find it useful as a shortcut to the Insure++ installation. Also, a tool may prompt you to set this environment variable. To set the PARASOFT environment variable correctly, you will need to know the name of the directory in which Insure++ has been installed on your system. Once you know this path you should define an environment variable called PARASOFT to be this pathname.

Typically, this can be performed by editing the file .cshrc in your home directory and adding a line similar to:

```
setenv PARASOFT <target directory>
```

For example: `setenv PARASOFT /usr/local/parasoft`

Step 8. Modify Your PATH

You must add the directory containing the executables to your execution path. Normally, you do this by adding to the definition of either the path or PATH variables, according to the shell you are using. The directory in which the executables are located will have a name that can be derived from the type of system you are running on, and is given to you by the install script.

A typical C-shell command would be:

```
set path=($path <target directory><arch>)
```

For example: `set path=($path /usr/local/parasoft/bin.linux2)`

If you are in doubt as to which directory to put on your search path, ask your system administrator for help.

Step 9. Modify Your Environment

After modifying the appropriate configuration files, you should execute the following commands to actually modify your working environment:

```
source ~/.cshrc
rehash
```

Step 10. Running An Example

Change to the Insure++ examples directory and run the makefile:

```
cd <target directory>/examples/c
make all
```

For example:

```
cd /usr/local/parasoft/examples/c
make all
```

Contacting Parasoft

Parasoft is committed to providing you with the best possible product support for Insure++. If you have any trouble using Insure++, please follow the procedure below in contacting our Quality Consulting department:

- Check the manual.
- Be prepared to recreate your problem.
- Know your Insure++ version. (You can find it by typing `insure` at the command prompt).
- Know your operating system version. (You can find it by typing `uname -a` at the command prompt).
- If the problem is not urgent, report it by e-mail or by fax.
- If you call, please use a phone near your computer. The Quality Consultant may need you to access Insure++ while you are on the phone.

Insure++ experts are available online to answer your questions.

Contact Information

	USA Headquarters	**Parasoft UK**
Phone	(888) 305-0041	+44 (020) 8263 2827
FAX	(626) 305-9048	+44 (020) 8263 2701
Email	Email: quality@parasoft.com	Email: quality@parasoft-uk.com
	Parasoft France	**Parasoft Germany**
Phone	+(33 1) 64 89 26 00	+49 7805/ 956 960
FAX	+(33 1) 64 89 26 10	+49 7805/ 919 714
Email	Email: quality@parasoft-fr.com	Email: quality@parasoft-de.com

Running Insure++

The goal of this section is to give you enough information to start compiling and running your own programs under Insure++. Then you should be able to start finding bugs in your own software.

You use Insure++ by:

1. Processing your program with the special `insure` program in place of your normal compiler. This creates a version of your code which includes calls to the Insure++ library and then passes it to your normal compiler.

   If you simply re-link your code, you will get a basic level of checking for heap corruption and errors in calls to common C functions as well as checking for memory leaks. If you wish to do comprehensive checking, you can recompile your code with Insure++ for the strongest possible runtime checking.

2. Running the program in the normal manner.

Note: On Linux x86, you can test your unmodified (uninstrumented) executable with Chaperon. This does not require any recompiling and relinking, or changing of environment variables. For more information see the section “Chaperon (Linux x86 Only)” on page 52.

During the compilation process, Insure++ detects and reports various errors including:

- Illegal typecasts
- Incorrect parameters specified to library routines
- Memory corruption errors which are "obvious" at compile time

During execution, Insure++ reports on a wide variety of programming errors. For an exhaustive description of the types of errors detected, see the section “Insure++” on page 33. For each error reported, you will see the source line that appears to be incorrect and an explanation of what type of error occurred.

Normally, Insure++ sends its output to `stderr`, but there is also a graphical tool for viewing error messages, Insra. For more information, see “Insra” on page 84.

The easiest way to learn how to use Insure++ is to use it on an example program and see what it does. This section introduces two of the examples supplied with Insure++: one C example and one C++ example. You may want to copy the appropriate files to a directory and perform the steps as they are described.

The examples chosen here illustrate some of the simpler features of Insure++ and have been chosen to help you start using the system quickly. Once you have gone through this section, you should be in a position to use Insure++ on your own programs.

Step-by-Step Integration

If you are working with a large application and don't want to jump right into recompiling all your code with Insure++, you can use the following method to integrate Insure++ into your development process step-by-step.

- Link your program with insure.
- Add the option insure++.summarize leaks to your `.psrc` file and run your program. Leaked blocks with stack traces will be reported in a summary report at the conclusion of your program.
- To find out when and where the blocks were leaked as well as find more bugs, start recompiling parts of your code with Insure++.

For an example of this procedure, see “Linking Leak With Insure++” on page 25.

A Simple C Example: Sorting

The C demonstration code used in this section is a very simple program which attempts to sort an array of numbers.

If you wish to follow along with the example in this section, you can save some typing time by using the source code supplied with Insure++.

To get started, make a directory for temporary use and copy the source code for this example to it with commands similar to

```
mkdir $HOME/insure
cd $HOME/insure
cp /usr/local/insure/examples/c/bubble1.c .
```

Compiling and Running Without Insure++

Once you've got a copy of the example program `bubble1.c` in your current directory, you can compile it with the command

```
cc -g -o bubble1 bubble1.c
```

and then run it from the shell in the normal manner.

```
bubble1
```

The program doesn't crash, and it doesn't print any error messages. Perhaps it's working?

Actually, this program has a serious error: a simple memory related bug.

Compiling Bubble1 With Insure++

Finding the error with Insure++ requires only that you recompile the program with the special `insure` command

```
insure -g -o bubble1 bubble1.c
```

`insure` simply replaces your normal compiler on the command line.

Note: The `-g` in both of the above commands is necessary on many platforms for Insure++ to be able to generate stack traces with file names and line numbers.

You can use the same options you normally use to compile and link your code by just replacing `cc` with `insure` in either your command lines or your makefile.

You may also compile the source code with compilers such as CC, gcc, or g++ with Insure instrumentation without modifying the `.psrc` file by doing the following: `insure <CC, gcc, or g++>`. For example, to compile `bubble1` in `CC` mode:

```
insure CC -g -o bubble1 bubble1.c
```

To compile `bubble1` in `gcc` mode:

```
insure gcc -g -o bubble1 bubble1.c
```

If you normally use a C compiler other than cc, you can make Insure++ do the same by creating a file called `.psrc` in your current directory and adding a line like `Insure++.compiler_c gcc`. This option tells Insure++ to use `gcc` instead of `cc` to compile C source files.

If your compiler is not directly supported by Insure++, you will also need to set the `compiler_acronym` option (see "Advanced Configuration Options Used by Insure++" on page 167 for more details). If you need to use the `compiler_acronym` option, you will also need to use the compiler option instead of the `compiler_c` or `compiler_cpp` options. Both the `compiler` and `compiler_acronym` options override any `compiler_c` or `compiler_cpp` options.

If you are using a makefile, things are often even easier, because many use the variable `CC` to define the name of the compiler to use. If this is true in your case, you can build a version of your program with Insure++ by typing the command

```
make CC=insure
```

You don't have to edit anything! Even better, you can build the original (unchecked) version or the Insure++ version by simply changing the command you type. If your makefile uses a different variable, e.g. `LD` , for the link command, you will need to use a command like

```
make CC=insure LD=insure.
```

Note: If your original LD definition is `/bin/ld`, replace it with `ins_ld`. If it was a compiler driver, such as `cc`, `gcc`, or `g++`, then replace it with `insure`.

Running Bubble1 With Insure++

Run the program `bubble1.c` just as you would if you hadn't compiled it with Insure++.

```
        bubble1
```

This time you get more interesting responses, as shown below:

```
[bubble1.c:27] **READ_BAD_INDEX**
>>             if(a[j-1] > a[j]) {

 Reading array out of range: a[j - 1]

 Index used : -1
 In block   : 0x0804b1a0 thru 0x0804b1c7 (40 bytes,
              10 elements)
              vector, declared at bubble1.c, 10

 Stack trace where the error occurred:
                     bubble_sort()  bubble1.c, 27
                            main()  bubble1.c, 16
```

The output from Insure++ indicates that something is wrong and indicates the exact line number where the error occurs.

The error detected is indicated by the error code `READ_BAD_INDEX` and occurs at line 20 of `bubble1.c`. The line of code that causes the error is also shown along with a description of the problem. The other information shown in the display is best understood by examining the source code for the example, shown below.

```
/*
 * File: bubble1.c
 */

/*
 * pad1 and pad2 reduce possibility of reading
 * garbage when a[-1] is erroneously dereferenced.
 */
int   pad1[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
int vector[10] = {4, 3, 6, 9, 1, 5, 8, 2, 0, 7};
int   pad2[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
main()
{
    bubble_sort(vector,
                sizeof(vector)/sizeof(vector[0]));
    exit(0);
}

bubble_sort(a, n)
    int a[], n;
{
    int i, j;

    for(i=0; i<n; i++) {
        for(j=0; j<n-i; j++) {
            if(a[j-1] > a[j]) {
                int temp;

                temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

We first declare an array which contains the list of values to be sorted in line 4. This is then passed from the main routine to the sorting subroutine in line 8. The remaining information presented in the Insure++ bug report can now be interpreted as follows.

- The illegal index used in line 20 has the value `-1`, which implies that the variable `j` must have the value `0`.
- The block of memory which is being accessed is fully described. Its starting and ending memory locations are given along with the size and number of elements in the array.
- The name of the array being accessed is given, including the location at which it was declared. Notice that this information describes the global variable `vector`, even though the `bubble_sort` routine is accessing this variable by the name `a`, as passed in its argument list.
- Finally, a stack trace is given which shows the sequence of function calls leading to this error.

From this information, it is hard to miss the cause of the problem in the code. The operation in line 20 is to compare an array element with its predecessor. This is the right operation to perform, but since we use the index values `j` and `j-1`, the loop range of `j` (in line 19) should start at `1`, not `0`.

This type of problem is very common and can easily go unnoticed in production code, because it doesn't crash the program and it may not even affect the result. In the example shown, the out-of-bounds value is quite likely zero, since it refers to another global variable.

Eliminating the Bug In Bubble1

The fix in this case is particularly simple - line 19 should actually read

```
19: for(j=1; j<n-i; j++) {
```

To see that this actually fixes the problem, either modify the source file or copy `bubble2.c` from the Insure++ examples directory (see "A Simple C Example: Sorting" on page 18). Compile and run it under Insure++ with the commands

```
insure -g -o bubble2 bubble2.c
bubble2
```

This time no errors are reported.

Using Insure++ With C++ Code

By default, Insure++ is set up to use the CC compiler with C++ source files on most platforms. The exceptions are Compaq Tru64 Unix (`cxx`), IBM AIX (`xlC`), and Linux (`g++`). If you use a different compiler, you need to insert the line

```
insure++.compiler_cpp [cxx|CC|g++|xlC]
```

into your `.psrc` file. This tells Insure++ to compile all C++ source files with the given compiler.

Another important consideration when using Insure++ is source code file extensions. When Insure++ sees a `.c` file, it automatically treats it as C code. Any file with a `.cc`, `.C`, `.ccp`, `.cxx` or `.c++` extension will be treated as C++ code. This is very important to understand. You cannot put C++ code in a file with a .c extension unless you also add the `insure++.c_as_cpp` on option to your `.psrc` file. For more information about this option, “Configuration Options” on page 162.

Linking C++ Objects With Insure++

If your makefile uses a separate link command with no source files on the link line, you must have the `insure++.compiler_default cpp` option in your `.psrc` file to tell Insure++ to use C++ linkage. If Insure++ only sees objects and libraries on the link line, it cannot tell whether the code is C or C++. By default, it assumes it is C code and uses C linkage. The above option changes the default to C++ linkage.

As an alternative to the above method, if you use only C++ code, you can set the compiler option in your `.psrc` file to your C++ compiler, e.g. compiler CC. This option overrides any `compiler_c`, `compiler_cpp`, or `compiler_default` options present and tells Insure++ to use the indicated compiler every time it is called, for both compiling and linking.

A C++ Example: Memory Leak

C++ can be a very difficult language in which to program, so we have significantly improved Insure++ to detect very hard-to-find bugs.

Often, code that contains serious errors can appear perfectly correct - at least until the problems start manifesting themselves in crashes, core dumps, or memory exhaustion. That is why we added capabilities like program tracing and detection of memory allocation conflicts, dead code, overloading operators and more.

To illustrate how Insure++ can detect tricky, well-disguised memory leaks in C++ code, let's consider the program whose source is presented below.

```
/*
 * File: leak.C
 */
#include <string.h>

union S1 {
        char *cp;
        S1() { cp=new char[10]; }
        S1(char *p) {
                cp=new char[10];
                strcpy(cp,p);
        }
        S1(S1 &s) {
                cp=new char[10];
                strcpy(cp,s.cp);
        }
        void mf(char *p) { strcpy(cp,p); }
};
void foo() {
        S1 s1,s2("Hello "),s3=s2;
        s1.mf("SADF");
        s3.mf("World");
}
int main() {
        foo();
        return(0);
}
```

Linking Leak With Insure++

Insure++ now detects memory leaks when the program is only linked with Insure++. Compiling and linking the example `leak.C` with the commands

```
g++ -g -c leak.C
insure -g -o leak leak.C
```

and executing

```
leak
```

with `insure++.summarize leaks` in your `.psrc` file activated produces the output shown below in the leak summary report for the program `leak`.

```
*************INSURE SUMMARY ****************** v6.1 **
*   Program      : leak             *
*   Arguments    : Not available    *
*   Directory    : /usr/local/parasoft/examples/cpp*
*   Compiled on  : Not available    *
*   Run on       : Jun 25, 2002  15:15:01  *
*   Elapsed time : 00:00:00         *
*   Malloc HWM   : 30 bytes         *
*****************************************************

MEMORY LEAK SUMMARY
===================

3 outstanding memory references for 30 bytes.

Leaks detected at exit
----------------------
        30 bytes      3 chunks allocated
                          malloc()  (interface)
                   __builtin_new()
                             foo()  leak.C, 20
                            main()  leak.C, 25
```

The output in tells us that there is a leak from each of the constructors in the `S1` class. In many cases this may be enough information to find and fix the bug. If it is not, however, Insure++ can give you more information, including where the leak actually occurred, not just where the leaked block was allocated.

Compiling and Running Leak With Insure++

Compiling and linking the (leak.C) example with the command

```
insure -g -o leak leak.C
```

and executing

```
leak
```

produces the output shown below.

```
[leak.C:23] **LEAK_SCOPE**
>> }

  Memory leaked leaving scope: cp

  Lost block : 0x0804bad0 thru 0x0804bad9 (10 bytes)
               cp, allocated at:
                          malloc()  (interface)
              ** routines compiled without debug info **
                          S1::S1()  leak.C, 8
                             foo()  leak.C, 20
                            main()  leak.C, 25

  Stack trace where the error occurred:
                             foo()  leak.C, 23
                            main()  leak.C, 25
```

The leak occurs because there is no destructor in `S1`. When `s1`, `s2`, and `s3` are called in `foo`, they appear to be on the stack, which would not cause memory to be allocated. However, `S1` calls new to allocate memory and does not have any way to deallocate it. This causes a large leak. Only the first leak is reported at runtime because by default Insure++ reports only one error per category per line. This behavior can be changed using the `insure++.report_limit .psrc` option.

Eliminating the Bug In Leak

This error can be easily corrected by adding a destructor to `S1`. For example, adding the following line of code between lines 16 and 17 would eliminate the bug.

```
~S1() { delete[] cp; } //destructor
```

Improving Insure++'s Compile-Time Performance

If you are compiling in a remotely mounted directory, one easy way to decrease Insure++'s compile time is to use the `temp_directory` option. This .psrc option controls where Insure++ writes its temporary files during compilation. If you use it to redirect temporary files to a local disk, compilation performance will improve dramatically. For example, adding the option

```
        insure++.temp_directory /tmp
```

to your `.psrc` file tells Insure++ to write its temporary files in the `/tmp` directory.

You can significantly speed up the execution of your program by using the `header_ignore` option in your .psrc file to avoid instrumenting header files that you know are correct. See “Configuration Options” on page 162 for more information about this option.

Chaperon Quick Test (Linux x86 Only)

If you want to check your code for runtime errors but do not have the time to instrument your code with Insure++, you can check your code with Chaperon. Chaperon mode is faster-- though slightly less thorough-- than regular (Source Code Instrumentation) mode. You can run your application in Chaperon mode by entering

```
Chaperon filename.exe
```

at the prompt. For a complete description of Chaperon, including examples, see “Chaperon (Linux x86 Only)” on page 52.

Maintaining Both Normal and Insure++ Builds

Another way to save time is to create a complete image of your project with Insure++ when you begin the debugging process. Then as you find and fix errors, you can just recompile one or two files at a time with

Insure++. This will cut down greatly on compilation time in comparison with recompiling every file every time you want to switch from a normal build to an Insure++ build, or vice versa.

The makefile shown below builds a program consisting of two source files, `func.c` and `main.c`. Typing make would build main in the current directory, using the default settings in the makefile. However, all that is necessary to build a completely separate version of the program with Insure++ is the command

```
        make CC=insure TDIR=insure
```

Alternatively, you can edit the makefile to redefine `CC` and `TDIR` each time you want to switch between a normal and an Insure++ build, if you prefer.

```
CC = cc
CFLAGS = -g
TDIR = .
OBJS = $(TDIR)/main.o $(TDIR)/func.o

$(TDIR)/main: $(OBJS)
        $(CC) $(CFLAGS) -o $(TDIR)/main $(OBJS)

$(TDIR)/main.o: main.c
        $(CC) $(CFLAGS) -o $(TDIR)/main.o -c main.c

$(TDIR)/func.o: func.c
        $(CC) $(CFLAGS) -o $(TDIR)/func.o -c func.c

clean:
        /bin/rm -rf $(TDIR)/*.o $(TDIR)/main
```

If you normally build libraries from your objects and do not add objects explicitly to your link line, you can do a similar trick by building a variable like `TDIR` into the object and library names, as shown in the makefile given in the figure below. In this case, a command like

```
        make CC=insure TARGET=_ins
```

would leave you with versions of your objects, libraries, and executable tagged with names ending in `_ins`.

```
CC = cc
CFLAGS = -g
TARGET =
OBJS = main$(TARGET).o
LIBS = libfunc$(TARGET).a

main$(TARGET): $(OBJS) $(LIBS)
         $(CC) $(CFLAGS) -o main$(TARGET) $(OBJS) $(LIBS)

libfunc$(TARGET).a: func$(TARGET).o
         ar ruv libfunc$(TARGET).a Func$(TARGET).o

main$(TARGET).o: main.c
         $(CC) $(CFLAGS) -o main$(TARGET).o -c main.c

func$(TARGET).o: func.c
         $(CC) $(CFLAGS) -o func$(TARGET).o -c func.c

clean:
         /bin/rm -rf *$(TARGET).o main$(TARGET)\
                                    libfunc$(TARGET).a
```

Common Insure++ Options

Insure++ is an extremely customizable tool. While this flexibility is one of the great strengths of Insure++, it can present a problem for the new user. Although we ship Insure++ with defaults that will serve the majority of users quite well, we realize that some users have their own special needs and preferences. To help you configure Insure++ for your use, we would like to suggest some of the most popular options used by Insure++ users over the years and explain what they do. You can then pick and choose those that will be helpful in your particular situation.

More information about all of the options is available in the section “Configuration Options” on page 162. All of the options listed there can be placed in a file called `.psrc` in your local build directory with a prefix of `insure++`. They are applicable at different times in the build process.

Comprehensive Testing

The programs `bubble2` and leak now run to completion, even when compiled with Insure++. Of course, `bubble1` and `leak` previously ran to completion, even though they contained the errors that Insure++ found. So what does it actually mean when Insure++ says there are no more errors? It means that Insure's testing is quite comprehensive. A program that passes Insure++ without any error messages will not contain any of the following:

- Uninitialized memory accesses
- Illegal pointer operations
- "Wild" pointer operations caused when a pointer skips from one data object to point at another
- Dynamic memory errors
- Accesses of memory blocks outside their legal bounds
- Memory leaks

Note that this is only a brief summary. The full set of errors detected by Insure++ is described in the section “Insure++” on page 33 and listed in the section “Error Codes” on page 197.

Preventing Errors With CodeWizard

You can run Insure++ with CodeWizard to perform both automatic error detection (Insure++) and automatic error prevention (CodeWizard) in a single step. CodeWizard (available separately from Parasoft) checks your code for design and coding problems that can lead to bugs and other problems later on. By finding and fixing problems early, you can save yourself untold amounts of debugging and maintenance time. In addition, you will be learning valuable coding techniques that can actually reduce the number of bugs in your code in the future.

By combining the analytical power of Code Wizard with the bug-finding prowess of Insure++, you can speed up your entire development process. At one glance, you'll see where the bugs are in your program and where trouble is likely to occur in the future. Fix all the errors now and you'll save yourself time and headaches later.

Download CodeWizard now!

See the CodeWizard Manual for more information on automatically preventing coding errors.

Optimizing Dynamic Memory With Inuse

Inuse is a graphical Insure++ add-on that allows you to watch how your programs handle memory in real-time. Inuse will help you to better understand the memory usage patterns of algorithms and how to optimize their behavior. With Inuse, you'll have a clear understanding of how your program actually uses (and abuses) memory. You can use Inuse to:

- See how much memory an application uses in response to particular user events
- Compare an application's overall memory usage to its expected memory usage
- Look for memory fragmentation to see if different allocation strategies might improve performance
- Detect the most subtle memory leaks, which can cause problems over time.

Running Inuse

In normal use, you should enter the `inuse` command once and simply leave it running as a background process.

```
inuse
```

Inuse will be run the next time Insure++ is run. For more information, see “Working With Inuse” on page 112.

Analyzing Code Coverage With TCA

The Total Coverage Analysis (TCA) add-on works hand-in-hand with Insure++ and reports which parts of your program have actually been tested by Insure++ and how often each block of code was executed. Using TCA with Insure++ can dramatically improve the efficiency of your testing and guarantee faster delivery of more reliable programs.

Running TCA

Insure++ tracks code coverage information in a file called `tca.log`. This file is located in the same directory that your executable was built into.

You can analyze Insure++ coverage from the command line or through the TCA GUI.

- To review code coverage information from the TCA GUI, click **File> Load** in the TCA window and select the `tca.log` file located in your program's directory.
- To review code coverage information from the command line, type: `tca tca.log`. For command line options, type: `tca`.

Note for Linux: TCA does not work with Chaperon. Since Chaperon works on the original executable (non-instrumented) and the `tca.map` is not even created, there will be no `tca.log`. For more information, see “Working With TCA” on page 138.

Insure++

Using Insure++ is easy. You simply recompile your program with Insure++ instead of your normal compiler. Running the program under Insure++ then generates a report whenever an error is detected; this report usually contains enough detail to track down and correct the problem.

Insure++ automatically detects errors that might otherwise go unnoticed in normal testing. Subtle memory corruption errors and dynamic memory problems often don't crash the program or cause it to give incorrect answers until the program is shipped to customers and they run it on *their* systems. Then the problems start.

Even if Insure++ doesn't find any problems in your programs, running it gives you the confidence that your program doesn't contain any errors.

Of course, Insure++ can't possibly check everything that your program does. However, its checking is extensive and covers every class of programming error. The following table lists the types of errors that Insure++ detects.

Memory Corruption	Pointer Abuse	Memory Leaks
Dynamic Memory Manipulation	Strings	Uninitialized Memory
Unused Variables	Data Representation Problems	Incompatible Variable Declarations
I/O Statements	Mismatched Arguments	Invalid Parameters In System Calls
Unexpected Errors In System Calls		

Memory Corruption

This is one of the most unpleasant errors that can occur, especially if it is well disguised. As an example of what can happen, consider the program shown below. This program concatenates the arguments given on the command line and prints the resulting string.

```
/*
 * File: hello.c
 */
#include <string.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;
    char str[16];

    str[0] = '\0';
    for(i=0; i<argc; i++) {
        strcat(str, argv[i]);
        if(i < (argc-1)) strcat(str, " ");
    }
    printf("You entered: %s\n", str);
    return (0);
}
```

If you compile and run this program with your normal compiler, you'll probably see nothing interesting. For example:

```
$ cc -g hello.c -o hello
$ ./hello
You entered :./hello
$ ./hello world
You entered: ./hello world
$ ./hello cruel world
You entered: ./hello cruel world
```

If this were the extent of your test procedures, you would probably conclude that this program works correctly, despite the fact that it has a very serious memory corruption bug.

If you compile with Insure++, the command `hello cruel world` generates the errors shown below, because the string that is being concatenated becomes longer than the 16 characters allocated in the declaration at line 7.

```
[hello.c:15] **WRITE_OVERFLOW**
>>         strcat(str, argv[i]);

  Writing overflows memory: <argument 1>

          bbbbbbbbbbbbbbbbbbbbbbbbbb
          |           16           | 2 |
          wwwwwwwwwwwwwwwwwwwwwwwwwwwwww

   Writing  (w) : 0xbfffeed0 thru 0xbfffeee1 (18 bytes)
   To block (b) : 0xbfffeed0 thru 0xbfffeedf (16 bytes)
                 str, declared at hello.c, 11

  Stack trace where the error occurred:
                          strcat()  (interface)
                            main()  hello.c, 15

**Memory corrupted.  Program may crash!!**

[hello.c:18] **READ_OVERFLOW**
>>     printf("You entered: %s\n", str);

  String is not null terminated within range: str

  Reading   : 0xbfffeed0
  From block: 0xbfffeed0 thru 0xbfffeedf (16 bytes)
             str, declared at hello.c, 11

  Stack trace where the error occurred:
                            main()  hello.c, 18

You entered: hello cruel world
```

Insure++ finds all problems related to overwriting memory or reading past the legal bounds of an object, regardless of whether it is allocated statically (that is, a global variable), locally on the stack, dynamically (with `malloc` or `new`), or even as a shared memory block.

Insure++ also detects situations where a pointer crosses from one block of memory into another and starts to overwrite memory there, even if the memory blocks are adjacent.

Pointer Abuse

Problems with pointers are among the most difficult encountered by C programmers. Insure++ detects pointer-related problems in the following categories

- Operations on `NULL` pointers.
- Operations on uninitialized pointers.
- Operations on pointers that don’t actually point to valid data.
- Operations which try to compare or otherwise relate pointers that don’t point at the same data object.
- Function calls through function pointers that don’t actually point to functions.

Below is the code for a second attempt at the “Hello world” program that uses dynamic memory allocation.

```
/*
 * File: hello2.c
 */
#include <stdlib.h>
#include <string.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    char *string, *string_so_far;
    int i, length;

    length = 0;

    for(i=0; i<argc; i++) {
        length += strlen(argv[i])+1;
        string = malloc(length+1);
/*
 * Copy the string built so far.
 */
        if(string_so_far != (char *)0)
            strcpy(string, string_so_far);
        else *string = '\0';

        strcat(string, argv[i]);
        if(i < argc-1) strcat(string, " ");
        string_so_far = string;
    }
    printf("You entered: %s\n", string_so_far);
    return (0);
}
```

The basic idea of this program is that we keep track of the current string size in the variable `length`. As each new argument is processed, we add its length to the `length` variable and allocate a block of memory of the new size. Notice that the code is careful to include the final `NULL` character when computing the string length (line 11) and also the space between strings (line 14). Both of these are easy mistakes to make. It's an interesting exercise to see how quickly Insure++ finds such an error.

The code in lines 19-24 either copies the argument to the buffer or appends it depending on whether or not this is the first pass round the loop. Finally in line 25 we point at the new, longer string by assigning the pointer `string` to the variable `string_so_far`.

If you compile and run this program under Insure++, you'll see "uninitialized pointer" errors reported for lines 19 and 20. This is because the variable `string_so_far` hasn't been set to anything before the first trip through the argument loop.

Memory Leaks

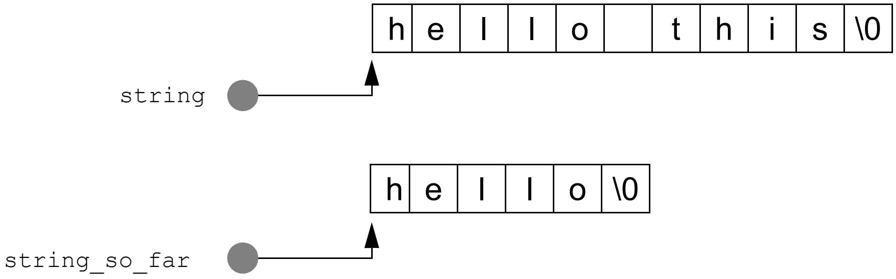
A “memory leak” occurs when a piece of dynamically allocated memory cannot be freed because the program no longer contains any pointers that point to the block. A simple example of this behavior can be seen by running the (corrected) “Hello world” program with the arguments

```
hello3 this is a test
```

If we examine the state of the program at line 27, just before executing the call to `malloc` for the second time, we observe:

- The variable `string_so_far` points to the string “`hello`” which it was assigned as a result of the previous loop iteration.
- The variable `string` points to the extended string “`hello this`” which was assigned on this loop iteration.

These assignments are shown schematically below; both variables point to blocks of dynamically allocated memory.

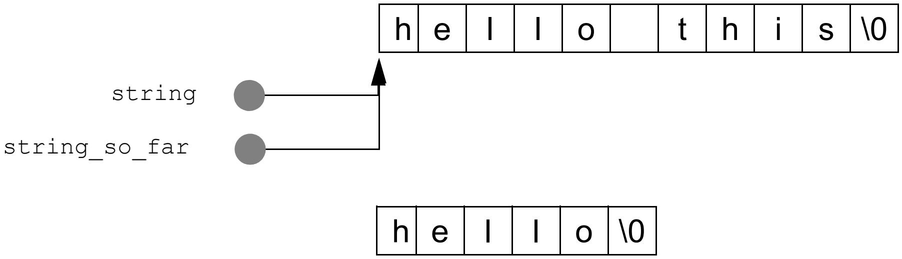


Pointer assignments before the memory leak

The next statement

```
          string_so_far = string;
```

will make both variables point to the longer memory block as shown below.



Pointer assignments after the memory leak

Once this happens, however, there is no remaining pointer that points to the shorter block. Even if you wanted to, there is no way that the memory that was previously pointed to by `string_so_far` can be reclaimed; it is permanently allocated. This is known as a “memory leak” and is diagnosed by Insure++ as shown below.

```
[hello3.c:28] **LEAK_ASSIGN**
>>          string_so_far = string;

  Memory leaked due to pointer reassignment: string

  Lost block : 0x0804bd68 thru 0x0804bd6f (8 bytes)
               string, allocated at hello3.c, 18
                          malloc()  (interface)
                            main()  hello3.c, 18

  Stack trace where the error occurred:
                            main()  hello3.c, 28
```

This example is called LEAK_ASSIGN by Insure++ since it is caused when a pointer is re-assigned. Other types of leaks that Insure++ detects include:

Leak Type	**Description**
LEAK_FREE	Occurs when you free a block of memory that contains pointers to other memory blocks. If there are no other pointers that point to these secondary blocks then they are permanently lost and will be reported by Insure++.
LEAK_RETURN	Occurs when a function returns a pointer to an allocated block of memory, but the returned value is ignored in the calling routine.
LEAK_SCOPE	Occurs when a function contains a local variable that points to a block of memory, but the function returns without saving the pointer in a global variable or passing it back to its caller.

Notice that Insure++ indicates the exact source line on which the problem occurs, which is a key issue in finding and fixing memory leaks. This is an extremely important feature, because it's easy to introduce subtle memory leaks into your applications, but very hard to find them all. Using Insure++, you can instantly pinpoint the line of source code which caused the leak.

Should Memory Leaks Be Fixed?

Whether or not this is a serious problem depends on your application. To get more information on the seriousness of the problem, add the "`Insure++ summarize leaks outstanding`" option to your `.psrc` file.

To get more information on the seriousness of the problem, check the **Summarize: Leaks** box in the **Reports** tab of the Insure++ Control Panel.

When you run the program again, you will see the same output as before, followed by a summary of all the memory leaks in your code.

```
MEMORY LEAK SUMMARY
===================

5 outstanding memory references for 78 bytes.

Leaks detected during execution
-------------------------------
55 bytes      4 chunks allocated at hello3.c, 18
                 malloc()  (interface)
                   main()  hello3.c, 18

Outstanding allocated memory
----------------------------
23 bytes 1 chunk allocated at hello3.c, 18
                 malloc()  (interface)
                   main()  hello3.c, 18
```

This shows that even this short program lost four different chunks of memory. The total of 78 bytes isn't very large and you might ignore it in a program this size. If this was a routine in a larger program, it would be a serious problem because every time the routine is called it allocates blocks of memory and loses some. As a result, the program gradually consumes more and more memory and will finally crash when the memory space on the host machine is exhausted.

This type of bug can be extremely hard to detect, because it might take literally days to show up. Insure++ only prints one error message although the summary indicates that four memory leaks occurred. This is because Insure++ normally shows only the first error of any given type at each particular source line. If you wish, you can change this behavior as described in “Displaying Repeated Errors” on page 70.

Note: A dynamically allocated memory block is categorized as a leak if a pointer to that block is lost during program execution. A block is categorized as outstanding memory if a pointer to the block is retained up to program termination, but the block is not freed prior to program termination.

Finding All Memory Leaks

For an even higher level of checking, we suggest the following algorithm for removing all memory leaks from your code.

1. Run your program from Inuse. If you see an increase in the heap size as you run the program, you are leaking memory.
2. Compile all source code, but not libraries, with Insure++. Clean all leaks that are detected by Insure++.
3. Compile everything that makes up your application with Insure++ -- source code and libraries. Clean any leaks detected by Insure++. If you do not have source for any of the libraries, skip this step and proceed to Step 4.
4. Examine each outstanding memory reference to determine whether or not it is a leak. If the pointer is passed into a library function, it may be saved. If this is the case, it is not a leak. Once every outstanding memory reference is understood, and those that are leaks are cleared, the program is free of memory leaks.

Dynamic Memory Manipulation

Using dynamically allocated memory properly is another tricky issue. In many cases, programs continue running well after a programming error causes serious memory corruption; sometimes they don't crash at all.

One common mistake is to try to reuse a pointer after it has already been freed. As an example we could modify the "Hello world" program to deallocate memory blocks before allocating the larger ones. Consider the following piece of code which does just that:

```
22:      if(string_so_far != (char *)0) {
23:              free(string_so_far);
24:              strcpy(string, string_so_far);
25:      }
26:      else *string = '\0';
```

If you run this code (`hello4.c`) through Insure++, you'll get another error message about a "dangling pointer" at line 23. The term "dangling pointer" is used to mean a pointer that doesn't point at a valid memory block anymore. In this case the block is freed at line 22 and then used in the following line. This is another common problem that often goes unnoticed, because many machines and compilers allow this particular behavior.

In addition to this error, Insure++ also detects the following errors:

- Reading from or writing to "dangling pointers."
- Passing "dangling pointers" as arguments to functions or returning them from functions.
- Freeing the same memory block multiple times.
- Attempting to free statically allocated memory.
- Freeing stack memory (local variables).
- Passing a pointer to `free` that doesn't point to the beginning of a memory block.
- Calls to `free` with `NULL` or uninitialized pointers.
- Passing non-sensical arguments or arguments of the wrong data type to `malloc`, `calloc`, `realloc` or `free`.

Another way that Insure++ can help you track down dynamic memory problems is through the `RETURN_FAILURE` error code. Normally, Insure++ will not issue an error if `malloc` returns a `NULL` pointer because it is out of memory. This behavior is the default, because it is assumed that the user program is already checking for, and handling, this case.

If your program appears to be failing due to an unchecked return code, you can enable the `RETURN_FAILURE` error message class (See “RETURN_FAILURE” on page 322). Insure++ will then print a message whenever any system call fails.

Strings

The standard C library string handling functions are a rich source of potential errors, since they do very little checking on the bounds of the objects being manipulated.

Insure++ detects problems such as overwriting the end of a buffer as described in “Memory Corruption” on page 34. Another common problem is caused by trying to work with strings that are not null-terminated, as in the following example:

```
/*
 * File: readovr2.c
 */
main()
{
    char junk;
    char b[8], c[8];
    strncpy(b, "This is a test",
                        sizeof(b));
    memset(c, 0, sizeof(c));
    printf("%s\n", b);
    return (0);
}
```

This program attempts to copy the string `This is a test` into a buffer which is only 8 characters long. Although it uses `strncpy` to avoid overwriting its buffer, the resulting copy doesn’t have a `NULL` on the end. Insure++ detects this problem in line 10 when the call to `printf` tries to print the string.

Uninitialized Memory

A particularly unpleasant problem to track down occurs when your program makes use of an uninitialized variable. These problems are often intermittent and can be particularly difficult to find using conventional means, since any alteration in the operation of the program may result in different behavior. It is not unusual for this type of bug to show up and then immediately disappear whenever you attempt to trace it.

Insure++ performs checking for uninitialized data in two sub-categories.

Category	Name	Description
1.	copy	Normally, Insure++ doesn't complain when you assign a variable using an uninitialized value, since many applications do this without error. In many cases the value is changed to something correct before being used, or may never be used at all.
2.	read	Insure++ generates an error report whenever you use an uninitialized variable in a context which cannot be correct, such as an expression evaluation.

To clarify the difference between these categories consider the following code.

```
/*
 * File: readuni1.c
 */
#include <stdio.h>

int main()
{
        struct rectangle {
                int width;
                int height;
        };

        struct rectangle box;
14:             int area;
15:
16:             box.width = 5;
17:             area = box.width*box.height;
18:             printf("area = %d\n", area);
19:             return (0);
20:     }
```

In line 17 the value of `box.height` is used to calculate a value which is invalid, since its value was never assigned. Insure++ detects this error in the `READ_UNINIT_MEM(read)`category. This category is enabled by default, so a message will be displayed.

If you changed line 17 to

```
17:             area = box.height;
```

Insure++ would report errors of type `READ_UNINIT_MEM(copy)` for both lines 17 and 18, but only if you had unsuppressed this error category.

Unused Variables

Insure++ can also detect variables that have no effect on the behavior of your application, either because they are never used, or because they are assigned values that are never used. In most cases these are not serious errors, since the offending statements can simply be removed, and so they are suppressed by default.

Occasionally, however, an unused variable may be a symptom of a logical program error, so you may wish to enable this checking periodically. See “UNUSED_VAR” on page 327 for more details.

Data Representation Problems

A lot of programs make either explicit or implicit assumptions about the various data types on which they operate. A common assumption made on workstations is that pointers and integers have the same number of bytes. While some of these problems can be detected during compilation, others hide operations with typecasts such as shown in the following example:

```
char *p;
int ip;

ip = (int)p;
```

On many systems this type of operation would be valid and would not cause any problems. However, when such code is ported to alternative architectures problems can arise. The code shown above would fail, for example, when executed on a PC (16-bit integer, 32-bit pointer) or a 64-bit architecture such as the Compaq Tru64 Unix (32-bit integer, 64-bit pointer).

In cases where such an operation loses information, Insure++ reports an error. On machines for which the data types have the same number of bits (or more), no error is reported.

Incompatible Variable Declarations

Insure++ detects inconsistent declarations of variables between source files. A common problem is caused when an object is declared as an array in one file:

```
int myblock[128];
```

but as a pointer in another:

```
extern int *myblock;
```

See the files `baddecl1.c` and `baddecl2.c` in the `examples` directory for an example. Insure++ also reports differences in size, so that an array declared as one size in one file and a different size in another will be detected.

I/O Statements

The `printf` and `scanf` family of functions are easy places to make mistakes which show up either as bugs or portability problems. For example, consider the following code:

```
foo()
{
        double f;

        scanf("%f", &f);
}
```

This code will not crash, but the value read into the variable `f` will not be correct, since its data type (`double`) doesn't match the format specified in the call to `scanf` (`float`). As a result, incorrect data will be transferred to the program.

In a similar way, the example `badform2.c`

```
foo()
{
        float f;

        scanf("%lf", &f);
}
```

corrupts memory, since too much data will be written over the supplied variable. This error can be very difficult to detect.

A more subtle issue arises when data types used in I/O statements "accidentally" match. The following code

```
foo()
{
        long l = 123;
        printf("l = %d\n", l);
}
```

functions correctly on machines where types `int` and `long` have the same number of bits, but fails otherwise. Insure++ detects this error, but classifies it differently from the previous cases. You can choose to ignore this type of problem while still seeing the previous bugs.

In addition to checking `printf` and `scanf` arguments, Insure++ also detects errors in other I/O statements. The code

```
foo(line)
        char line[80];
{
        gets(line);
}
```

works as long as the input supplied by the user is shorter than 80 characters, but fails on longer input. Insure++ checks for this case and reports an error if necessary.

Note: This case is somewhat tricky, since Insure++ can only check for an overflow after the data has been read. In extreme cases the act of reading the data will crash the program before Insure++ gets the chance to report it.

Mismatched Arguments

Calling functions with incorrect arguments is a common problem in many programs, and can often go unnoticed. For example, Insure++ detects the error in the following program

```
double foo(dd)
        double dd;
{
        return dd + 1.0;
}

main()
{
        printf("Result = %f\n", foo(1));
}
```

in which the argument passed to the function `foo` in `main` is an integer rather than a floating point number.

Note: Converting this program to ANSI style (for example, with a function prototype for `foo`) makes it correct since the argument passed in `main` will be automatically converted to `double`. Insure++ doesn't report an error in this case.

Insure++ detects several different categories of errors, which you can enable or suppress separately depending on which types of bugs you consider important.

- Sign errors - Arguments agree in type but one is signed and the other unsigned (for example, `int` vs. `unsigned int`).
- Compatible types - The arguments are different data types which happen to occupy the same amount of memory on the current machine (for example, `int` vs. `long` if both are 32-bits). While this error might not cause problems on your current machine, it is a portability problem.
- Incompatible types - Similar to the example above. Data types are fundamentally different or require different amounts of memory. `int` vs. `long` would appear in this category on machines where they require different numbers of bits.

Invalid Parameters In System Calls

Interfacing to library software is often tricky, because passing an incorrect argument to a routine might cause it to fail in an unpredictable manner. Debugging such problems is much harder than correcting your own code, since you typically have much less information about how the library routine should work.

Insure++ has built-in knowledge of a large number of system calls and checks the arguments you pass to ensure correct data type and, if appropriate, correct range.

For example, the code:

```
void myrewind(FILE fp)
{
     fseek(fp, (long)0, 3);
}
```

would generate an error since the last argument passed to the `fseek` function is outside the legal range.

Unexpected Errors In System Calls

Checking the return codes from system calls and dealing correctly with all the error cases that can arise is a very difficult task. Very rarely will a program deal with all possible cases correctly.

An unfortunate consequence of this is that programs can fail unexpectedly because some system call fails in a way that had not been anticipated. The consequences of this can range from a nasty "core dump" to a system that performs erratically at the customer location.

Insure++ has a special error class, `RETURN_FAILURE`, that can be used to detect these problems. All the system calls known to Insure++ contain special error checking code that detects failures. Normally these errors are suppressed, since it is assumed that the application is handling them itself, but they can be enabled at runtime by unsuppressing `RETURN_FAILURE` in the Suppressions Control Panel. Any system call that returns an error code will then print a message indicating the name of the routine, the arguments supplied, and the reason for the error.

This capability detects *any* error in *any* known system call. Among the potential benefits are automatic detection of errors in the following situations:

- `malloc` runs out of memory.
- Files that do not exist.
- Incorrectly set permission flags.
- Incorrect use of I/O routines.
- Exceeding the limit on open files.
- Inter-process communication and shared memory errors.
- Unexpected "interrupted system call" errors.

Chaperon (Linux x86 Only)

Chaperon checks all data memory references made by a process, whether in the developer's compiled code, language support routines, shared or archive libraries, or operating system kernel calls. Chaperon detects and reports reads of uninitialized memory, reads or writes that are not within the bounds of allocated blocks, and allocation errors such as memory leaks.

Chaperon works with existing executable programs. In most cases, Chaperon requires no recompilation and no relinking, and no changes to environment variables. Just add `Chaperon` to the beginning of the command line; Chaperon will run the process and check all data memory references.

When Chaperon detects improper behavior, it issues an error message identifying the kind of error and where it occurred. Improper behavior is any access to a logically unallocated region, a Read (or Modify) access to bytes which have been allocated but not yet Written, or attempts to free the same block twice.

Chaperon also detects memory blocks that have been allocated and not freed. If such a block is not reachable by starting from the stack, or from statically allocated regions, and proceeding through already reached allocated blocks, then the block is a “memory leak.” Such a block cannot be freed without some oracle to specify its address as the parameter to `free()`. At `exit()` Chaperon will report leaked and outstanding memory blocks if the corresponding `Insure++.summarize` option is in effect:

```
Chaperon <program_name> <arguments>
```

Using Chaperon does not require running under a debugger, but Chaperon also works with existing debuggers such as gdb. For more information, see “Using Chaperon With gdb” on page 63.

Requirements and Limitations

- ELF format executables and shared libraries, with /lib/ld-linux.so.2 -> ld-2.1.1.so (or compatible), as the ELF program interpreter. The important interfaces are _dl_runtime_resolve, _dl_relocate_object _dl_debug_state, and _r_debug.
- Any x86 processor [x >= 3] running Linux. In case of opcode conflict between manufacturers, Chaperon follows the Intel documentation.
- Linux kernel 2.4.x, 2.2.5, or compatible. Other kernels will work with adjustment of the accounting for system calls.
- malloc/free/etc must not call sigaction that gets used.
- vfork() is remapped to fork(). Programs depending on vfork() semantics may not work properly.
- 32-bit code (no 0x67 address size prefix; but 0x66 operand size prefix is OK), flat model. Any explicit cs, ds, or ss segment selector in the instruction stream must equal the corresponding current actual selector. Chaperon's access accounting treats all offsets as belonging to segment ds. Application code using es, fs, or gs does run; but the accounting may become confused.

Bitfields

Chaperon accounts for memory on a byte-by-byte basis. Since the mapping between bytes and bitfields need not be 1-to-1 and onto, there are problems. By default, Chaperon uses heuristics to guess that some instruction sequences (that would otherwise generate complaints of Read before Write) correspond to legitimate bitfield operations, and the heuristics enable Chaperon to suppress those complaints.

They also cause a whole byte to be marked as Written as soon as the first write to any bitfield that intersects it. The heuristics are not complete; there will still be "false positive" complaints of Read before Write. Some source statements and expressions that are not bitfields can generate code that looks like bitfields, and for which the heuristics should be disabled; use the command line parameter -bitfields=0.

The general palliative for cleaning up Chaperon complaints about bitfields is to clear all words that contain bitfields as soon as the memory is allocated, perhaps using `memset` and perhaps employing a `union`. This can even be more efficient, but some programmers consider it to be distasteful.

Symbols, Tracebacks, and Compilers

Chaperon's underlying execution engine and tracking for memory state depend only on x86 architecture, and are compiler independent. But the generation and reporting of tracebacks and symbols relies on Chaperon being able to find and identify the code and symbols. Chaperon recognizes the functions involved in dynamic binding (`.dynsym` symbols) and static binding (`.symtab` symbols).

System Calls

Chaperon checks the documented memory access behavior of kernel calls for Linux 2.2.x and 2.4.x except `bdflush`, `capget`, `capset`, `getpmsg`, `ipc` (but `shm*` shared memory calls *are* checked), `modify_ldt`, `nfsservctl`, `prctl`, `putpmsg`, `quotactl`, `sysfs`, and `vm86*`. Chaperon understands the “regular” `SYS_ioctl` calls whose command word uses `_IOR`, `_IOW`, or `_IOWR`, plus important non-regular cases such as `TIOC*` (terminal control) and `SIOC*` (socket control).

Space

Chaperon runs in the same execution context and address space as the process that Chaperon is checking. The linear coefficients of space overhead are 2 bits of accounting info per byte of address space used by the application, plus (`16 + 8*traceback_length`) bytes per active allocated block. Process sizes greater than about 500MB have not been well explored.

Memory States and Access Accounting

State	Read or Modify Access	Write Access
Unallocated	Error: Read before Allocate	Error: Write before Allocate
Allocated but not Written	error: Read before Write	OK: becomes Allocated and Written
Allocated and Written	OK	OK

Allocators: `malloc`, `calloc`, `realloc`, `memalign`, `__libc_malloc`, `__libc_calloc`, `__libc_realloc`, `__libc_memalign`, stack growth (push, create frame), `__brk`, `brk`, `__sbrk`, `sbrk`, `mmap`

De-allocators: `free`, `realloc`, `__libc_free`, `__libc_realloc`, `__brk`, `brk`, `__sbrk`, `sbrk`, stack trim (pop, delete frame), `munmap`

Other known functions: `memcpy`, `memset`, `memmove`, `memchr`, `bcopy`, `bzero`, `strcat`, `strchr`, `__stpcpy`, `strcpy`, `strrchr`. These are optimized for faster performance, and/or to reduce the clutter of multiple error messages that arise from a single call, and/or to suppress “false positive” Read before Write messages from instruction sequences that are known to be used to implement write-allocate cache control, or speculative word-wide reading of byte arrays.

Handling of `realloc(ptr, size)`:

```
If 0==size then free(ptr);
else if 0==ptr then mlloc(size);
else {free(ptr); malloc(size)}
```

and arrange for the new contents of the `malloc()`ed region to equal the old contents for the first min(old_size, new_size) bytes. See also “Bitfields” on page 53.

Examples

Note: The numeric values of addresses might not match when the examples are re-run. For instance, locations in the stack (`0xbfffffff` and lesser nearby locations) depend on the number of characters in environment variables. Locations in shared libraries (`0x40000000` and greater nearby locations) change with different versions and different values of `LD_PRELOAD`. Locations in application code (`0x08040000` and greater nearby locations) depend on compiler and compiler options.

WRITE_OVERFLOW

To run this example, first, compile the code with gcc.

```
gcc -g -o writover writover.c
```

Run the new executable on Chaperon.

```
Chaperon writover
```

Chaperon should report errors such as:

```
/*
 * $RCSfile: writover.c,v $
 * $Revision: 32.2 $
 *
 * Comments:
 *
 * (C) Copyright Parasoft Corporation 1998.
 * All rights reserved.
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF
 * Parasoft. The copyright notice above does not
 * evidence any actual or intended publication of such
 * source code.
 *
 */

// writover.c
#include <stdlib.h>

int
main()
{
        /* An example of WRITE_OVERFLOW*/
        char *p = malloc(10);
        p[11] = 3;
        return 0;
}

$Chaperon ./writover

// Chaperon(tm) memory access checker version 2.0
// 2002-06-18.
// Copyright 1999 BitWagon Software LLC.  All
// rights reserved.
// Copyright 2001 Parasoft Corp. All rights reserved.
[writover.c:22] (Thread 0) **WRITE_OVERFLOW**
>>         p[11] = 3;

  Writing overflows memory.

          bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
          |              10              | 1 | 1 |
                                             wwwww

   Writing  (w) : 0x0804966b thru 0x0804966b (1 byte)
   To block (b) : 0x08049660 thru 0x08049669 (10 bytes)
                 block allocated at writover.c, 21
                            main()  writover.c, 21

  Stack trace where the error occurred:
                            main()  writover.c, 22

**Memory corrupted.  Program may crash!!**


Exit with return code 0  (0x0000).
  16  total blocks allocated
  0  total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.
```

READ_UNINIT_MEM

```
/*
 * $RCSfile: readunin.c,v $
 * $Revision: 32.3 $
 *
 * Comments:
 *
 * (C) Copyright Parasoft Corporation 1998.
 * All rights reserved.
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF
 * Parasoft
 * The copyright notice above does not evidence any
 * actual or intended publication of such source code.
 *
 */

// readunit.c
int
main()
{
        int x, y;
        if (3==x) {
                return y+2;
        }
        else {
                return y-1;
        }
}

$ Chaperon readunin
// Chaperon(tm) memory access checker version 2.0
// 2002-06-18.
// Copyright 1999 BitWagon Software LLC.  All rights
// reserved.
// Copyright 2001 Parasoft Corp. All rights reserved.
[readunin.c:19] (Thread 0) **READ_UNINIT_MEM(read)**
>>         if (3==x) {

  Reading uninitialized memory.

  Pointer : 0xbfffee84
  Stack trace where the error occurred:
                            main()  readunin.c, 19

[readunin.c:23] (Thread 0) **READ_UNINIT_MEM(read)**
>>                 return y-1;

  Reading uninitialized memory.

  Pointer : 0xbfffee80

  Stack trace where the error occurred:
                            main()  readunin.c, 23


Exit with return code 1075315610  (0x4018039a).
  15  total blocks allocated
  0  total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.
```

FREE_DANGLING

```
/*
 * $RCSfile: freedngl.c,v $
 * $Revision: 32.2 $
 *
 * Comments:
 *
 * (C) Copyright Parasoft Corporation 1998.  All rights
 * reserved.
 * THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF
 * Parasoft
 * The copyright notice above does not evidence any
 * actual or intended publication of such source code.
 *
 */

// freedngl.c
#include <stdlib.h>

int
```

```
main()
{
        char *p = malloc(13);
        free(p);
        free(p);
        return 0;
}

$Chaperon freedngl
// Chaperon(tm) memory access checker version 2.0
// 2002-06-18.
// Copyright 1999 BitWagon Software LLC.  All rights
// reserved.
// Copyright 2001 Parasoft Corp. All rights reserved.
[freedngl.c:22] (Thread 0) **FREE_DANGLING**
>>         free(p);

  Freeing dangling pointer.

  Pointer : 0x080496a8
  In block: 0x080496a8 thru 0x080496b4 (13 bytes)
                 block allocated at freedngl.c, 20
                           main()  freedngl.c, 20

stack trace where memory was freed:
                           main()  freedngl.c, 21

  Stack trace where the error occurred:
                           main()  freedngl.c, 22

**Memory corrupted.  Program may crash!!**


Exit with return code 0  (0x0000).
  16  total blocks allocated
  1  total blocks freed.
Chaperon searching memory blocks...
```

Summarize Leaks

Make sure that your `.psrc` file has the following line in it:

```
insure++.summarize leaks outstanding
```

Then run Chaperon again.

```
// Chaperon(tm) memory access checker version 2.0
// 2002-06-18.
// Copyright 1999 BitWagon Software LLC.  All rights
// reserved.
// Copyright 2001 Parasoft Corp. All rights reserved.
[writover.c:22] (Thread 0) **WRITE_OVERFLOW**
>>         p[11] = 3;

  Writing overflows memory.

          bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
          |              10              | 1 | 1 |
                                             wwwww

   Writing  (w) : 0x0804966b thru 0x0804966b (1 byte)
   To block (b) : 0x08049660 thru 0x08049669 (10 bytes)
                 block allocated at writover.c, 21
                            main()  writover.c, 21

  Stack trace where the error occurred:
                            main()  writover.c, 22

**Memory corrupted.  Program may crash!!**


Exit with return code 0  (0x0000).
  16  total blocks allocated
  0  total blocks freed.
Chaperon searching memory blocks...
```

```
End of memory leak processing.
************************** INSURE SUMMARY****** v6.1 **
*   Program      : ./writover                        *
*   Arguments    : Not available                     *
*   Directory    : Not available                     *
*   Compiled on  : Not available                     *
*   Run on       : Jun 26, 2002  16:12:32            *
*   Elapsed time : 00:00:00                          *
*   Malloc HWM   : 2269 bytes (2K)                   *
******************************************************

MEMORY LEAK SUMMARY
===================

1 outstanding memory reference for 10 bytes.

Leaks detected at exit
----------------------
        10 bytes 1 chunk allocated at writover.c, 21
                            main()  writover.c, 21


PROBLEM SUMMARY - by type
===============

Problem                 Reported      Suppressed
-------------------------------------------------
WRITE_OVERFLOW               1                0
-------------------------------------------------
TOTAL                        1                0
-------------------------------------------------

PROBLEM SUMMARY - by location
===============

WRITE_OVERFLOW: Writing overflows memory, 1 unique
  occurrence
         1 at writover.c, 22
```

Using Chaperon With gdb

Chaperon can be used with your existing gdb, or with a modified gdb-5.0, provided by Parasoft in `INSTALLDIR/bin.linux2/gdb.exe`.

Either version can be used to set a breakpoint in `_Insure_trap_error`, which allows you to stop program execution at a point where a memory reference error is detected, and examine program state, values of variables, etc.

The Parasoft version of gdb could also be used to set breakpoints in your binary and execute gdb commands, such as `next`, `step`, `continue`.

Note: Your existing gdb will be able to set breakpoints in your executable as well, but you will not be able to properly continue execution after the breakpoint.

For example:

```
$gdb.exe Chaperon
GNU gdb 5.0  extended 2000-09-12 by Parasoft Corporation for
Chaperon on Linux x86
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.
Type "show warranty" for details.
This GDB was configured as "i586-pc-linux-gnu"...
(gdb) b gdb_setup
Breakpoint 1 at 0x1700: file chap0.S, line 62.
(gdb) r chaptest
Couldnt get a file descriptor referring to the console
// Chaperon(tm) memory access checker version 2.0
// 2002-06-26.
// Copyright 1999 BitWagon Software LLC.  All rights
// reserved.
// Copyright 2002 Parasoft Corp. All rights reserved.
```

```
Breakpoint 1, gdb_setup () at chap0.S:62
62  Copyright 1999 BitWagon Software LLC.  All rights
reserved.
Current language:  auto; currently asm
(gdb) b main
Breakpoint 2 at 0x80484d6: file chaptest.c, line 23.
(gdb) disable 1
(gdb) c

Breakpoint 2, main () at chaptest.c:23
23          return foo();
Current language:  auto; currently c
(gdb) b _Insure_trap_error
Breakpoint 3 at 0x1706: file chap0.S, line 76.
(gdb) c
[chaptest.c:6] (Thread 0) **READ_UNINIT_MEM(read)**
>>     if (p[1]) {

  Reading uninitialized memory.

  Pointer : 0x080496c1
  In block: 0x080496c0 thru 0x080496d3 (20 bytes)
                  block allocated at chaptest.c, 4
                            func()  chaptest.c, 4
                             foo()  chaptest.c, 18
                            main()  chaptest.c, 23

  Stack trace where the error occurred:
                            func()  chaptest.c, 6
                             foo()  chaptest.c, 18
                            main()  chaptest.c, 23
```

```
Breakpoint 3, _Insure_trap_error () at chap0.S:76
76       .stabs "gdb_setup:F(0,1)",N_FUN,0,62,gdb_setup
Current language:  auto; currently asm
(gdb) where
#0  _Insure_trap_error () at chap0.S:76
#1  0x804847c in func () at chaptest.c:6
#2  0x80484bb in foo () at chaptest.c:18
#3  0x80484db in main () at chaptest.c:23
#4  0x401bc5b0 in __libc_start_main () from /lib/libc.so.6
(gdb) quit

Exit with return code 146  (0x0092).
  16  total blocks allocated
  0  total blocks freed.
Exit 1
```

Finally, you can find the gdb-5.0.patch file, which contains diffs against the original gdb-5.0 source in the Insure++ distribution directory.

Reports

The error reports that have already been shown indicate that Insure++ provides a great deal of information about the problems encountered in your programs. Insure++ also provides many ways of customizing the presentation of this information to suit your needs.

Default Behavior

By default, Insure++ adopts the following error reporting strategy:

- Error messages are “coded” by a single word shown in uppercase, such as `READ_OVERFLOW`, `LEAK_SCOPE`, and so on.
- Messages about error conditions are displayed unless they have been suppressed by default or in a site specific configuration file. See “Error Codes” on page 197 for a list of error condition messages.
- Only the first occurrence of a particular (unsuppressed) error at any given source line is shown. See “Report Summaries” on page 77 for ways to change this behavior.
- Error messages are sent to the console (`stderr)`, the Insra GUI, or to a separate report file. See “The Report File” on page 67 or “Sending Messages To Insra” on page 91.
- Each error shows a stack trace of the previous routines, displayed all the way back to your `main` program.

The Report File

Normally, error reports are displayed on the UNIX `stderr` I/O stream. Users interested in sending output to Insra should consult the section “Insra” on page 84. If you wish to send both your program’s output and the Insure++ reports to a file, you can use the normal console redirection method. An alternative is to have Insure++ redirect only its output directly by adding an option similar to

```
insure++.report_file bugs.dat
```

to your `.psrc` file. This tells Insure++ to write its reports to the file bugs.dat, while allowing your program's output to display as it normally would. Whenever this option is in effect you will see a "report banner" similar to

```
** Insure++ messages will be written to bugs.dat **
```

on your terminal when your program starts to remind you that error messages are being redirected. To suppress the display of this banner add the option

```
insure++.report_banner off
```

to your `.psrc` file.

Normally the report file is overwritten each time your program executes, but you can force messages to be appended to an existing file with the command

```
insure++.report_overwrite false
```

If you want to keep track of the reports from multiple runs of your code, an alternative is to have Insure++ automatically generate filenames for you based on a template that you provide. This template takes the form of a string of characters with tokens such as `%d`, `%p`, or `%V` embedded in the template. Each of these is expanded to indicate a certain property of your program as indicated in the section “Configuration Options” on page 162.

For example, the advanced configuration :

```
report_file %v-errs.%D
```

when executed with a program called `foo` at 10:30 a.m. on the 21st of December 2001, might generate a report file with the name

```
foo-errs.20011221103032
```

The last two digits are the seconds after 10:30 on which execution began.

Note: Programs which `fork` will automatically have a `-%n` added to their format strings unless a `%n` or `%p` token is explicitly added to the format string by the user. This ensures that output from different processes will always end up in different report files.

You can also include environment variables in these filenames so that

```
$HOME/reports/%v-errs.%D
```

generates the same filename as the previous example, but also ensures that the output is placed in the reports sub-directory of the user's `HOME`.

This method is very useful for keeping track of program runs during development to see how things are progressing as time goes on.

Customizing the Output Format

By default, Insure++ displays a particular banner for each error report, which contains the filename and line number containing the error, and the error category found. For example:

```
[foo.c:10] **READ_UNINIT_MEM(copy)**
```

If you wish, you can modify this format to suit either your aesthetic tastes or for some other purpose, such as enabling the editor in your integrated environment to search for the correct file and line number for each error.

Customization of this output is achieved by setting the `error_format` option in your .psrc file to a string of characters containing embedded tokens which represent the various pieces of information that you might wish to view. See “Advanced Configuration Options Used by Insure++” on page 167 for more information.

For example, the command

```
error_format "\"%f\", line %l: %c"
```

would generate errors in the following format:

```
"foo.c", line 8: READ_UNINIT_MEM(copy)
```

which is a form recognized by editors such as GNU Emacs.

Note: Notice how the embedded double quote characters require backslashes to prevent them being interpreted as the end of the format string.

A multi-line format can also be generated with a command such as

```
error_format "%f, line %l\n\t%c"
```

which might generate errors in the following format:

```
foo.c, line 8
        READ_UNINIT_MEM(copy)
```

Displaying Process Information

When using Insure++ with programs which `fork` into multiple processes, you might wish to display additional process-related information in your error reports. For example, adding the option

```
insure++.error_format
"%f, line %l: \n\tprocess %p@%h: %c"
```

in your `.psrc` file would generate errors in the form

```
foo.c, line 8:
process 1184@gobi: READ_UNINIT_MEM(copy)
```

which contains the name of the machine on which the process is running and its process ID.

Displaying the Time At Which the Error Occurred

It is often convenient to know exactly when various errors occurred. You can extend the error reports generated by Insure++ in this fashion by adding the `%d` and/or `%t` characters to the error report format as specified in your `.psrc` file. For example, the format

```
insure++.error_format "%f:%l, %d %t <%c>"
```

generates error reports in the form

```
foo.c:8, 12-Jun-2002 14:24:03 <READ_NULL>
```

Displaying Repeated Errors

The default configuration suppresses all but the first error of any given kind at a source line. You can display more errors by modifying the `.psrc` file in either your working or `HOME` directory.

For example, adding the line

```
        insure++.report_limit 5
```

to your `.psrc` file will show the first five errors of each type at each source line.

Setting the value to zero suppresses any messages except those shown in summaries (see “Report Summaries” on page 77).

Setting the `report_limit` value to -1 shows all errors as they occur.

Note: Not all information is lost by showing only the first (or first few) errors at any source line. If you enable the report summary you will see the total number of each error at each source line.

Limiting the Number of Errors

If your program is generating too many errors for convenient analysis, you can arrange for it to exit (with a non-zero exit code) after displaying a certain number of errors by adding the line

```
        insure++.exit_on_error number
```

to your `.psrc` file and re-running the program. After the indicated number of errors, the program will exit. If number is less than or equal to zero, all errors are displayed.

Changing Stack Traces

There are two potential modifications you can make to alter the appearance of the stack tracing information presented by Insure++ to indicate the location of an error.

By default, Insure++ will read your program's symbol table at start-up time to get enough information to generate stack traces. To get file and line information, you will need to compile your programs with debugging information turned on (typically via the `-g` switch). If this is a problem, Insure++ can generate its own stack traces for files compiled with Insure++. You can select this mode by adding the options

```
insure++.symbol_table off
insure++.stack_internal on
```

to your `.psrc` file. The `stack_internal` option will take effect after you recompile your program, while the `symbol_table` option can be toggled at runtime. In this case, the stack trace will display

```
** routines not compiled with insure **
```

in place of the stack trace for routines which were not compiled with Insure++. This will also make your program run faster, particularly at start-up, since the symbol table will not be read.

If your program has routines which are deeply nested, you may see very long stack traces. You can reduce the amount of stack tracing information made available by adding an option like

```
insure++.stack_limit 4
```

into your `.psrc` file. If you run your program again, you will see at most the last four levels of the stack trace with each error. **Note:** Chaperon does not support this option.

The value "0" is valid and effectively disables tracing.

The value "-1" is the default and indicates that the full stack trace should be displayed, regardless of length.

Stack traces are also presented to show the function calling sequence when blocks of dynamically allocated memory were allocated and freed. In a manner similar to the `stack_limit` option, the `malloc_trace` and `free_trace` options control how extensive these stack traces are.

Searching For Source Code

Normally, Insure++ remembers the directory in which each source file was compiled and looks there when trying to display lines of source code in error messages. Occasionally your source code will no longer exist in this directory, possibly because of some sophisticated “build” or “make” process.

You can give Insure++ an alternative list of directories to search for source code by adding a value such as

```
        source_path .:/users/boswell/src:/src
```

to the .psrc file in your current working or HOME directories.

The list can contain any number of directories separated by colons.

Note: Insure++’s error messages normally indicate the line of source code responsible for a problem on the second line of an error report, after the >> mark. If this line is missing from the report, it means that the source code could not be found at runtime.

Suppressing Error Messages

The previous sections described issues which can affect the appearance of particular error messages. Another alternative is to completely suppress error messages of a given type which you either cannot or do not want to correct.

The simplest way of achieving this is to add lines similar to

```
insure++.suppress EXPR_NULL, PARM_DANGLING
```

to your `.psrc` file and re-run the program. No suppressed error messages will be displayed, although they will still be counted and displayed in the report summary (see “The Bugs Summary” on page 78).

In this context, certain wild-cards can be applied so that, for instance, you can suppress all memory leak messages with the command

```
insure++.suppress LEAK_*
```

You can suppress all errors with the command

```
insure++.suppress *
```

which has the effect of only creating an error summary. If the error code has sub-categories, you can disable them explicitly by listing the sub-category codes in parentheses after the name. For example:

```
insure++.suppress BAD_FORMAT(sign, compatible)
```

Alternatively,

```
insure++.suppress BAD_FORMAT
```

suppresses all sub-categories of the specified error class.

Suppressing Error Messages By Context

In addition to suppressing and unsuppressing errors by category or file, you can also suppress and unsuppress error messages by context. For example, to suppress `READ_NULL` errors occurring in routines with names beginning with the characters `sub`, enter:

```
insure++.suppress READ_NULL {  sub*   *  }
```

The interpretation of this syntax is as follows:

- The stack context is enclosed by a pair of braces.
- Routine names can either appear in full or can contain the `*` or `?` wildcard characters. The former matches any string, while the latter matches any single character.
- An entry consisting of a single `*` character matches any number of functions, with any names.
- Entries in the stack context are read from left to right with the left-most entries appearing lowest (or most recently) in the call stack.

With these rules in mind, the previous entry is read as:

- The lowest function in the stack trace (that is, the function generating the error message) must have a name that begins with `sub` followed by any number of other characters.
- Any number of functions of any name may appear higher in the function call stack.

A rather drastic, but common, action is to suppress any errors generated from within calls to the X Window System libraries. If we assume that these functions have names which begin with either "X" or "_X", we could achieve this goal with the statements

```
insure++.suppress all { * X* * }
insure++.suppress all { * _X* * }
```

which suppresses errors in any function (or its descendents) which begins with either of the two sequences.

As a final example, consider a case in which we are only interested in errors generated from the routine `foobar` or its descendents. In this case, we can combine `suppress` and `unsuppress` commands as follows

```
insure++.suppress all
insure++.unsuppress all { * foobar * }
```

Note: Error suppression is only possible for functions that appear in stack traces which list error locations. For example, consider the following error report for `READ_DANGLING`:

```
block committed at:
>       f2()
>       f1()
>stack trace where memory was decommitted:
>       f3()
>       e::g()
>stack trace where the error occurred:
>       g2()
>       g1()
```

You may suppress either of the functions `g1()` or `g2()` where the error occurred by using

```
Suppress READ_DANGLING { * g1 * }
```

or

```
Suppress READ_DANGLING { * g2 * }
```

or

```
Suppress READ_DANGLING { g*  * }
```

In this instance, however, you may not suppress either of the functions in the stack trace where memory was decommitted.

Suppressing Messages by File/Line

In addition to suppressions based on stack traces, you can suppress error messages based on the file/line generating the message.

The syntax for this type of suppression is:

```
file:line#
in file
```

Examples:

```
suppress readbadindex at foo.h:32
```

This suppresses readbadindex error messages at line 32 of `foo.h` at both compile time and runtime.

```
suppress parserwarning in header.h
```

This suppresses all parser warnings in `header.h`.

Wildcards are not supported in filenames for this syntax. However, this syntax can be used at both compile time and runtime (unlike stack trace suppressions, which can only be used at runtime).

It is illegal to have both a stack trace suppression and a file/line suppression on the same line. For example:

```
suppress myerror {a b c} at foo.c:3
```

Suppressing Other Warning Messages

For other compile time warning messages that do not have an associated number, there is another suppress option available. The `suppress_output` option takes a string as an argument and will suppress any message that includes text which matches the string. For example:

```
insure++.suppress_output wrong arguments passed
```

would suppress the warning from the previous section, as well as any others that included this text string.

Enabling Error Messages

Normally, you will be most interested in suppressing error messages about which you can or want to do nothing. Occasionally, you will want to enable one of the options that is currently suppressed, either by system default or one of your own `.psrc` files“Error Codes” on page 197 for more information.This is achieved by adding a line similar to the following to your `.psrc` file:

```
insure++.unsuppress RETURN_FAILURE
```

in the **Item** field. **Unsupress** follows the same rules as **Suppress**. For more information, see the “Suppressions Control Panel” on page 24.

Report Summaries

Normally, you will see error messages for individual errors as your program proceeds. Using the other options described so far, you can enable or disable these errors or control the exact number seen at each source line. This technique is most often used to systematically track down each problem, one by one.

However, it is often useful to obtain a summary of the problems remaining in a piece of code in order to track its progress. Insure++ supports the following types of summary reports:

- A bug summary which lists all outstanding bugs according to their error codes.
- A leak summary which lists all memory leaks - that is, places where memory is being permanently lost.
- An outstanding summary which lists all outstanding memory blocks - that is, places where memory is not being freed, but is not leaked because a valid pointer to the block still exists.
- A coverage summary which indicates how much of the application’s code has been executed.

Note: None of these options are displayed by default.

The Bugs Summary

This report summary is enabled by adding the option

```
insure++. summarize bugs
```

to your `.psrc` file and re-running your program.

In addition to the normal error reports, you will then also see a summary such as the one shown below.

```
******************* INSURE SUMMARY *********v6.1*****
*   Program      : hello
*   Arguments    : this is bug summary test
*   Directory    : /home/Insure++/examples/c
*   Compiled on  : Jun 25, 2002  15:22:58
*   Run on       : Jun 26, 2002  13:16:43
*   Elapsed time : 00:00:00
*   Malloc HWM   : 0 bytes
******************************************************
PROBLEM SUMMARY - by type
===============

          Problem                Reported      Suppressed
          -------------------------------------------------
          READ_OVERFLOW                3                0
          WRITE_OVERFLOW               2                1
          -------------------------------------------------
          TOTAL                        5                1
          -------------------------------------------------

PROBLEM SUMMARY - by location
===============

READ_OVERFLOW: Reading overflows memory, 3 unique occur-
rences
         1 at hello.c, 15
         1 at hello.c, 16
         1 at hello.c, 18

WRITE_OVERFLOW: Writing overflows memory, 2 unique occur-
rences
         2 at hello.c, 15
         1 at hello.c, 16
```

The first section is a header which indicates the following information about the program being executed.

- The name of the program.
- Any command line arguments, if available.
- The directory from which the program was run.
- The time the program was compiled.
- The time the program was executed.
- The length of time needed to execute the program.

This information is provided so that test runs can be compared accurately as to the arguments and directory of test. The time and date information is supplied to correlate with bug tracking software.

The second section gives a summary of problems detected according to the error code and frequency. The first numeric column indicates the number of errors detected but not suppressed. This is the total number of errors, which might differ from the number reported, since, by default, only the first error of any particular type is reported at each source line. The second column indicates the number of bugs which were not displayed at all due to `suppress` commands.

The third section gives details of the information presented in the second section, broken down into source files and line numbers.

The Leak Summaries

The simplest memory leak summary is enabled by adding the line

```
insure++.summarize leaks outstanding
```

to your `.psrc` file and re-running your program.

The output indicates the memory (mis)use of the program, as shown below.

```
********************* INSURE SUMMARY ********* v6.1 **
*   Program      : leakscop
*   Arguments    :
*   Directory    : /home/Insure++/examples/c
*   Compiled on  : Jun 26, 2002  13:15:27
*   Run on       : Jun 26, 2002  13:17:54
*   Elapsed time : 00:00:00
*   Malloc HWM   : 10 bytes
******************************************************

PROBLEM SUMMARY - by type
===============

          Problem                Reported      Suppressed
          -------------------------------------------------
          LEAK_SCOPE                   1                0
          -------------------------------------------------
          TOTAL                        1                0
          -------------------------------------------------
```

```
PROBLEM SUMMARY - by location
===============

LEAK_SCOPE: Memory leaked leaving scope, 1 unique occurrence
         1 at leakscop.c, 10

MEMORY LEAK SUMMARY
===================

1 outstanding memory reference for 10 bytes.

Leaks detected during execution
-------------------------------
        10 bytes 1 chunk allocated at leakscop.c, 9
                          malloc()  (interface)
                           gimme()  leakscop.c, 9
                            main()  leakscop.c, 15
```

The first section summarizes the “memory leaks” which were detected during program execution, while the second lists leaked blocks that were detected at program exit. These are potentially serious errors, in that they typically represent continuously increasing use of system resources. If the program is “leaking” memory, it is likely to eventually exhaust the system resources and will probably crash.

The first number displayed is the total amount of memory lost at the indicated source line, and the second is the number of chunks of memory lost. Note that multiple chunks *of different sizes* may be lost at the same source line - depending on which options you are using.

To customize the report, there are three options available:

1. `leak_combine`

   The `leak_combine` option controls how Insure++ merges information about multiple blocks. The default behavior is to combine all information about leaks which were allocated from locations with identical stack traces (`leak_combine trace`). It may be that you would rather combine all leaks based only on the file and line they were allocated, independent of the stack trace leading to that allocation. In that case, you would use `leak_combine location`. Or, you may simply want one entry for each leak (`leak_combine none`).

2. `leak_sort`

   The `leak_sort` option controls how the leaks are sorted after having been combined. The options are `none`, `location`, `trace`, `size`, and `frequency` (`size` is the default). Sorting by `size` lets you look at the biggest sources of leaks, sorting by `frequency` lets you look at the most often occurring source of leaks, and sorting by `location` provides an easy way to examine *all* your leaks.

3. `leak_trace`

   The `leak_trace` option causes a full stack trace of each allocation to be printed, in addition to the actual file and line where the allocation occurred.

The third section shows the blocks which are allocated to the program at its termination and which have valid pointers to them. Since the pointers allow the blocks to still be freed by the program (even though they are not), these blocks are not actually leaked. This section is only displayed if the `outstanding` keyword is used. Normally, these blocks do not cause problems, since the operating system will reclaim them when the program terminates. However, if your program is intended to run for extended periods, these blocks are potentially more serious.

The Coverage Summary

The coverage summary is enabled by adding the line

```
insure++.summarize coverage
```

to your `.psrc` file and re-running your program.

In addition to the normal error reports, you will see a summary indicating how much of the application's source code has been tested. The exact form of the output is controlled by the `.psrc` file option `coverage_switches`, which specifies the command line switches passed to the `tca` command to create the output.

If this variable is not set, it defaults to

```
insure++.coverage_switches tca -dS
```

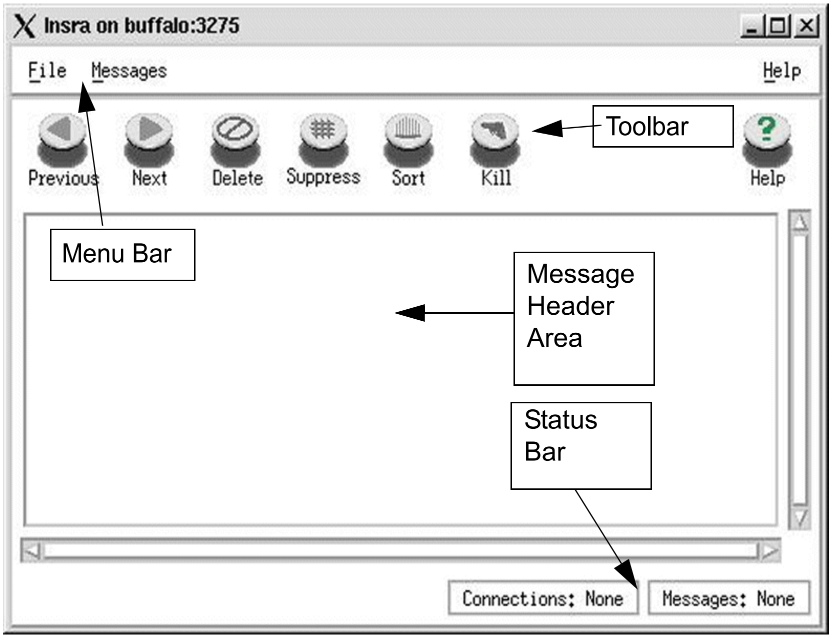
which displays an application level summary of the test coverage such as

```
COVERAGE SUMMARY
================
     0 blocks untested
    28 blocks tested

100% covered
```

Insra

Insra is a Graphical User Interface (GUI) for displaying error messages generated by Insure++. The messages are summarized in a convenient display, which allows you to quickly navigate through the list of bug reports and violation messages, suppress messages, invoke an editor for immediate corrections to the source code, and delete messages as bugs are fixed.



The Insra GUI

The Insra GUI contains the following components:

- **Menu Bar:** Contains options for manipulating files, messages, and finding online help regarding the Insra GUI.
- **Toolbar:** Contains options for viewing, navigating through, and suppressing messages.
- **Message Header Area:** Contains session headers and message headers for programs currently connected to Insra.
- **Status Bar:** Reports the number of error messages currently displayed and the number of active connections.

The ensuing subsections contain more information on these GUI components.

Menu Bar

The menu bar contains commands that manage Inuse memory functions. Available commands are detailed below:

File

- **Load...** a file for inspection through Insure++.
- **Save** the currently open file.
- Save a file under a new name and location using **Save as...**
- Save a file in ASCII (text) format using **Save ASCII**
- Save a file in ASCII (text) format using **Save As ASCII...**
- Choose an executable file and **Run** it.
- **Exit** the Insra GUI.

Messages

- **Prev** or **Next** to choose among messages.
- **Delete** to remove a message from the list.
- **Clear All** to clear all messages from the list.

- **Sort** to sort among the messages in the list.
- **Suppress** to suppress selected messages in the list.
- **Debug** to activate the Visual C++ window and start debugging the process within it to the point of execution where an error has occurred.

Help

- **Overview: About Insra** to access online help concerning the Insra GUI.
- **Insra Toolbar** to access online help concerning the Insra GUI toolbar.
- **Sending Messages to Insra** to access online help concerning sending messages to Insra.
- **Suppressing Messages** to access online help concerning suppressing messages in Insra.
- **Viewing Source Files** to access online help concerning viewing source files from the Insra GUI.
- **Troubleshooting** to access online help concerning troubleshooting tips for the Insra GUI.
- **About** to display which version of Insra you are running, as well as Parasoft contact information.

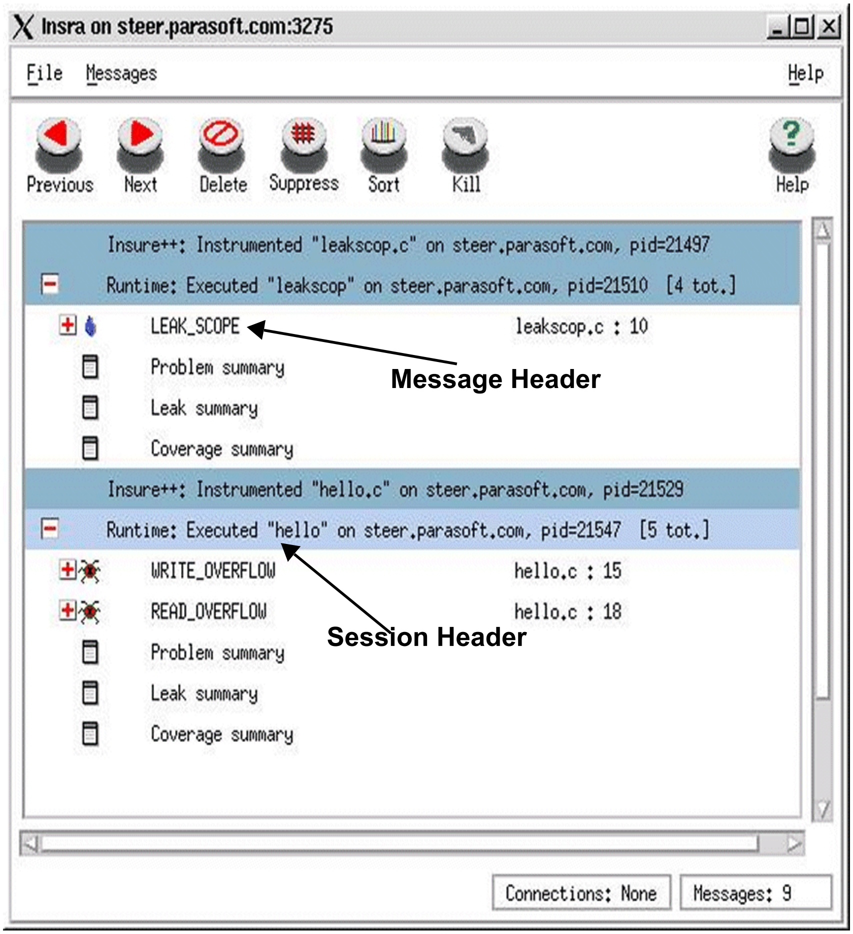
Toolbar

The toolbar allows you to:

- Scroll through using the **Previous** or **Next** buttons.
- **Delete** selected messages as bugs are fixed.
- **Suppress** errors detected by Insure++.
- **Sort** messages by order (time) reported, error category, or directory and file.
- **Kill** the selected active connection.
- Access online **Help**.

The Message Header Area

The message header area contains session headers and message headers for programs currently connected to Insra, as shown in the following graphic:



```
        Insure++: Instrumented "leakscop.c" on steer.parasoft.com, pid=21497
   ⊟    Runtime: Executed "leakscop" on steer.parasoft.com, pid=21510  [4 tot.]
     ⊞      LEAK_SCOPE                            leakscop.c : 10
            Problem summary
            Leak summary
            Coverage summary
        Insure++: Instrumented "hello.c" on steer.parasoft.com, pid=21529
   ⊟    Runtime: Executed "hello" on steer.parasoft.com, pid=21547  [5 tot.]
     ⊞      WRITE_OVERFLOW                        hello.c : 15
     ⊞      READ_OVERFLOW                         hello.c : 18
            Problem summary
            Leak summary
            Coverage summary
```

Connections: None | Messages: 9

Session Header

When the first error is detected for a particular compilation or execution, a session header is sent to Insra. The session header includes the following information:

- Compilation/execution
- Source file/program
- Host on which the process is running
- Process ID

Message Header

There are several types of message headers. Messages generated by Insure++ include:

- Error Category. For example: `LEAK_SCOPE`.
- File name
- Line number

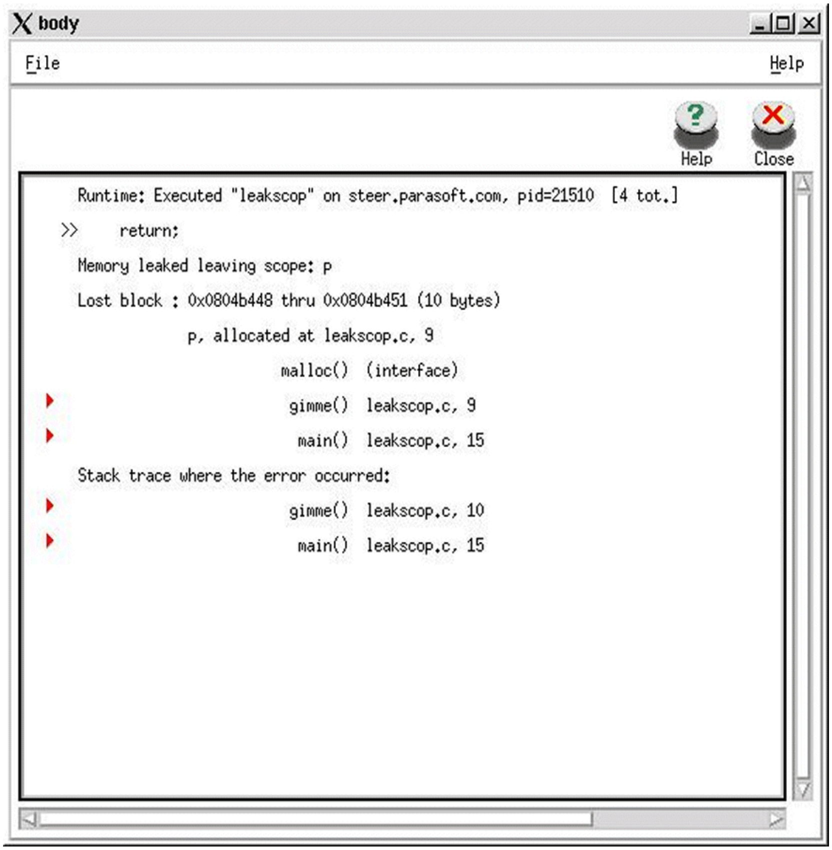
Message headers will also appear for various summary reports generated by Insure++. These reports are generated using the `.psrc` options. See "Report Summaries" on page 77 for more information. Double-clicking on a message header will open up the message window for the error or summary report selected.

The Status Bar

During compilation/runtime, Insure++ makes a connection to Insra each time an error is detected. The status bar reports the number of error messages currently displayed and the number of active connections. An active connection is denoted by a yellow star to the left of the session header. A connection remains active as long as the program is compiling/running. Insra will not allow you to delete a session header as long as its connection remains active, and you may not exit Insra until all connections have been closed.

Message Window

The message window opens when you double-click on a message header. This window contains the error message or summary report for the selected message header, as shown in the following graphic:



```
   Runtime: Executed "leakscop" on steer.parasoft.com, pid=21510  [4 tot.]
 >>    return;
   Memory leaked leaving scope: p
   Lost block : 0x0804b448 thru 0x0804b451 (10 bytes)
                p, allocated at leakscop.c, 9
                          malloc()  (interface)
                           gimme()  leakscop.c, 9
                            main()  leakscop.c, 15
   Stack trace where the error occurred:
                           gimme()  leakscop.c, 10
                            main()  leakscop.c, 15
```

Error Message

Insure++ error messages include:

- Line of source code where the error occurred.
- Explanation of the error detected.
- Stack traces for quick reference to the original source.

The stack traces are “live” and can be clicked once to launch an editor for viewing and correcting the indicated line of code. For more information, see “Viewing Source Files” on page 95.

All messages sent to Insra are marked with a special icon. Refer to the following table for a brief description of each icon.

Icon	Explanation
🐞	Insure++ error message
📄	Insure++ summary report
💧	Memory leak
⚡	Caught exception

Sending Messages To Insra

By default, all Insure++ output is sent to stderr. To redirect messages to Insra, simply add the following line to your `.psrc` file.

```
insure++.report_file insra
```

This will redirect both compile-time and run-time messages to Insra.

The option

```
insure++.runtime.report_file insra
```

will send only runtime messages to Insra. Compile-time messages will continue to be sent to `stderr`.

The option

```
insure++.compile.report_file insra
```

will send only compile-time messages to Insra. Runtime messages will continue to be sent to `stderr`.

With `insure++.report_file insra` in your `.psrc` file, each time an error is detected, Insure++ attempts to establish a connection to Insra. If Insra is not yet running, it will automatically start. Once the connection is established, a session header and all corresponding message headers will be reported in the order they were detected. Each new compilation or program, with its own session header and messages, will be displayed in the order in which it connected to Insra.

Viewing and Navigating

Message headers sent to Insra are denoted by a specific icon. For more information, see “The Insra GUI” on page 85. The body of the currently selected message is displayed in a separate message window. Double-click the message header to view the message itself. The message header area and the message window are both resizable, and scroll bars are also available to access text that is not visible. Currently active messages become inactive when they are deleted or suppressed.

Selecting An Editor

In addition to the location of the source file, Insra must also know the name of your editor and the command line syntax in order to display the correct file and line from the original source code.

Insra obtains this information by reading the `.psrc` option value

```
insra.visual [editor_command]
```

This value may contain the special tokens `%f` and `%l`, which represent the file name and line number, respectively.

The command will then be executed to load the file into your editor. It is most important to include the full path of any binary that lives in a location not pointed to by your `PATH` environment variable. If the variable has not been set, `vi` will be used by default.

Some editors are not X applications and must be run in a terminal window. `vi` requires the following command in order to lead the file successfully:

```
insra.visual xterm -e vi +%l %f
```

Other editors (for example, Emacs) do not require an external terminal program like xterm when configured for use as an X application. In this case, the command string should be similar to the following:

```
insra.visual emacs +%l %f
```

Note: Most implementations of vi and Emacs appear to be sensitive to the order of the line number and file name command line arguments, requiring the line number to precede the file name.

Deleting Messages

Once error messages have been read and analyzed, the user may wish to clear them from the window. The **Delete** button on the Insra toolbar allows you to remove error messages from the display as errors are corrected in your code. A message or an entire session may be removed from the display by selecting an entry in the message header area and clicking the **Delete** button. A message can also be deleted by selecting **Messages> Delete** from the menu bar.

Suppressing Messages

You can easily suppress (turn "off") error messages which you do not want Insure++ to generate. The Suppressions window allows you to insert, modify, and delete suppression options for Insure++ error messages. The suppression options you choose can be saved into your `.psrc` files so that they will be used again the next time you use Insure++.

Note: To access the Suppressions window, click the **Suppress** button in the Insra toolbar.

The Suppressions Window Toolbar

Moving from left to right across the toolbar in the Suppressions Window:

- The **Previous** and **Next** arrows select the next higher or next lower suppression option, respectively.
- The **Up** and **Down** arrows move the currently selected suppression option up or down in the order of options. Because Insure++ follows suppression options from top to bottom, the order in which they are listed affects the outcome of the suppression. (For more information, see “Reports” on page 66.)
- The **Delete** button deletes the currently selected suppression option.
- The **Insert** button inserts a new suppression option below the currently selected option. If you had an error message selected when you pressed the **Suppress** button on the Insra GUI, a suppression option for that particular error message will be inserted. Otherwise, the default suppression option (suppress all error messages) will be inserted. Suppression options are easy to edit. To change an option, simply follow the directions given below.
- The **Save** button writes all the suppression options which have been marked as persistent (see below) into the indicated `.psrc` files (see below). These suppression options will be in effect the next time you use Insure++.
- The **Help** button provides context-sensitive help, which in this window means that clicking anywhere will bring up this file.
- The **Close** button closes the suppression window.

Editing Suppression Options

An individual suppression option consists of five parts, listed below from left to right:

- **Suppress**/**Unsuppress**: This field specifies whether the error message listed is to be suppressed (as indicated by a speaker with an X through it) or unsuppressed (indicated by a speaker with no X through it). Double-clicking on the field toggles between suppressing and unsuppressing the error message.
- **Persistence**: This field specifies whether the suppression option will be saved to the `.psrc` file under which it is listed. Double-clicking this field toggles it from persistent (the field is checked) to temporary (the field is unchecked). Options marked as persistent will be added to the appropriate `.psrc` files when the save button is clicked. Options marked as temporary will be discarded. Options with an X in this field cannot be made persistent, either because they are hardwired or because the file in which it is placed is not writable. New options are marked as persistent by default.
- **Item**: Double-clicking this field allows you to type in the name of the error message you would like to suppress or unsuppress. You can use a wildcard (`*`) to match all error messages and also suppress and unsuppress by error category and context. For more information on suppressing error messages, see “Suppressing Error Messages” on page 73.
- **File**: Double-clicking this field allows you to type in the file for which you would like to suppress or unsuppress messages. Entering a blank field will insert a `*` which will match all files.
- **Note**: You may use this field to enter your own notes regarding the suppression option listed.

Configuration (.psrc) Files

The headers in the window show the various locations in which .psrc files reside. Insra will display the suppression options as read from each file under the appropriate header. When you add a new option using the Insert button, it will be inserted below the currently selected option. You can then move it into the file in which you would like it saved, or mark it as temporary by double-clicking the persistence field (see above).

There are two special locations where suppression options may reside other than actual .psrc files: hardwired options and command line options. The former are set internally by Insure++, and therefore cannot be permanently changed. They can be edited and/or removed in the window temporarily, however. The latter are options passed using the -Zop and -Zoi options on the Insure++ command line. These options, like hardwired options, cannot be made persistent, but can be moved into a .psrc file if you decide that you want to make them permanent.

The Kill Process

When an active connection is selected, pressing the **Kill** button will stop the selected compilation or execution.

Viewing Source Files

You can view the corresponding source file and line number for a particular error message by double clicking any line of the stack trace displayed in the message window. In most cases, the file and line number associated with a given message have been transmitted to Insra. If Insra is unable to locate the source file, a dialog box will appear requesting that you indicate the correct source file.

Saving/Loading Messages To A File

All current messages can be saved to a file by selecting **File> Save** or **File> Save As** from the menu bar. A dialog box allows you to select the destination directory and name of the report file. Report files have the default extension `rpt`. After a report file name has been selected, subsequent **File> Save** selections save all current messages into the report file without prompting for a new filename. A previously saved report file can be loaded by selecting **File> Load** from the menu bar. A dialog box then allows you to select which report file to load.

Help

On-line help can be obtained by choosing **Help** from the menu bar. This provides a list of topics on the use of Insra.

Setting Preferences

You can modify Insra's appearance with `.psrc` configuration options.

These options are:

`insra.body_background_color [White|color]`
Specifies the color used for the message body area background. The default is white.

`insra.body_font [Fixed|font]`
Specifies the font used for the message body text. The default is fixed.

`insra.body_height [number of rows]`
Specifies the starting height of the message window in number of rows of visible text. The default is 8.

`insra.body_text_color [Black|color]`
Specifies the color used for the message body text. The default is black.

```
insra.body_width [columns of text]
```

Specifies the starting width of the message window in number of columns of visible text. The default is 80, but if this value is set to a different value than header_width, then the larger value will be used.

```
insra.button_style [Round|square]
```

Specifies the shape of buttons that will be shown on toolbars. The default is round.

```
insra.coloured_shadows [on|off]
```

Specifies if round button shadows will be re-colored with the color of the application background. The default is on.

```
insra.expose_on_message [on|off]
```

Specifies if the Insra GUI will be placed on top of windows stack if it receives a new message. The default is off. (This option works only in by-time view mode.)

```
insra.follow_messages [on|off]
```

Specifies if the main window messages area will be automatically scrolled to follow arriving messages. The default is off. Note: this option works only in by-time view mode.

```
insra.header_background_color [White|color]
```

Specifies the color used for the message header area background. The default color is white.

```
insra.header_font [Fixed|font]
```

Specifies the font used for the message header text. The default is fixed.

```
insra.header_height [number of rows]
```

Specifies the starting height of the message header in number of rows of visible text. The default is 8.

`insra.header_highlight_color [LightSteelBlue2|color]`
Specifies the color used to indicate the currently selected message or session header in the message header area. The default is LightSteelBlue2.

`insra.header_highlight_text_color [Black|color]`
Specifies the color used for the text of the currently selected message of session header in the message header area. The default is black.

`insra.header_session_color [LightSkyBlue3|color]`
Specifies the color used for session header text. The default is LightSkyBlue3.

`insra.header_session_text_color [Black|color]`
Specifies the color used for session header text. The default is black.

`insra.header_text_color [Black|color]`
Specifies the color used for message header text. The default color is black.

`insra.header_width [number of columns]`
Specifies the starting width of the header area in number of columns of visible text. The default is 80, but if this value is set to a different value than body_width, the larger value will be used.

`insra.mark_unique [on|off]`
Specifies if messages not duplicated across all connections from the same tool has to be marked by special arrow-like icon. The default is on.

`insra.port [port_number]`
Specifies which port Insra should use to communicate with Insure++ compiled programs. The default is 3255.

`insra.sourcepath [dir_path1 dir_path2 ...]`

Specifies directories to be searched by Insra to find source files launching editor or showing source lines.

`insra.toolbar [on|off]`

Specifies whether Insra's toolbar is displayed. All toolbar commands can be chosen from the menu bar. The default is on.

`insra.viewmode [by_error|by_file|by_time|off|tool]`

Specifies initial view mode. Allows user to override view mode settings made by tools connecting with Insra.

- `by_error` - Insra GUI is initially set in view-by-error-category mode
- `by_file` - Insra GUI is initially set in view-by-file mode
- `by_time` - Insra GUI is initially set in view-by-time mode
- `off` - Insra GUI is launched with the default view mode (i.e. by-time mode)
- `tool` - means that view mode will be set by tool that first connects with the Insra GUI. This is the default setting.

`insra.visual [editor command]`

Specifies how Insra should call an editor to display the line of source code causing the error. Insra will match the `%l` token to the line number and the `%f` token to the file name before executing the command. It is important to include the full path of any binary that lives in a location not on your path. Setting this option with no command string disables source browsing from Insra. The default is `xterm -e vi +%l %f`

Troubleshooting

The following sections detail the most common errors encountered when using Insra. If you still encounter trouble after trying one of the following solutions, or if you encounter a different symptom from those listed below, contact the Parasoft Quality Consulting department. See “Contacting Parasoft” on page 15 for more information.

Insra Does Not Start Automatically

Symptom:While compiling or running, your program seems to hang when error output is directed to Insra and Insra is not yet running.

Solution: Run Insra by hand. Type

```
        insra &
```

at the prompt, wait for the Insra window to appear and then run or compile your program again. Output should now be sent to Insra.

Multiple Insra Users On One Machine

Symptom: When more than one user is attempting to send message reports to Insra, messages are lost.

Solution: Each invocation of Insra requires a unique port number. By default, Insra uses port 3255. If collisions are experienced — for example, multiple users on one machine — set the `.psrc` option insra.port to a different port above 1024. Ports less than 1024 are officially reserved for suid-root programs and should not be used with Insra.

Source Browsing Is Not Working

Symptom:

```
***Error while attempting to spawn browser execvp failed!
```

Solution: Insra attempted to launch your editor to view the selected source file, but could not locate your editor on your path. Make sure that this application is in a directory that is on your path or that you call it with its complete pathname.

Selective Checking

By default, Insure++ will check for bugs for the entire duration of your program. If you are only interested in a portion of your code, you can make some simple, unobtrusive changes to the original source to achieve this.

When you compile with `insure`, the pre-processor symbol `__INSURE__` is automatically defined. This allows you to conditionally insert calls to enable and disable runtime checks.

For example, assume that you are not interested in events occurring during the execution of a hypothetical function `grind_away`. To disable checking during this function, you can modify the code as shown below:

```
void grind_away() {
#ifdef __INSURE__
         _Insure_checking_enable(0);
                  //disables Insure++ checking
#endif
              ... code ...

#ifdef __INSURE__
         _Insure_checking_enable(1);
                  //enables Insure++ checking
#endif
}
```

Now when you compile and run your program, it will not check for bugs between the calls to `_Insure_checking_enable`.

If you do not want to modify the code for the `grind_away` function itself, you can add calls to `_Insure_checking_enable` around the calls to `grind_away`.

Every call to disable checking should be balanced by a call to enable checking. You should therefore be wary of using this function together with exceptions, such as `longjmp`, and so on.

The Insure++ runtime library will continue to record memory allocations and deallocations while checking is disabled. Thus, disabling checking does not affect the runtime library's knowledge of a program's memory usage.

Interacting with Debuggers

While it is our intent that the error messages generated by Insure++ will be sufficient to identify most programming problems, it will sometimes be useful to have direct access to the information known to Insure++. This can be useful in the following situations:

- You are running your program from a debugger and would like to cause a breakpoint whenever Insure++ discovers a problem.
- You are tracing an error using the debugger and would like to monitor what Insure++ knows about your code.
- You wish to add calls to your program to periodically check the status of some data.

Available Functions

Whenever Insure++ detects an error, it prints a diagnostic message and then calls the routine `_Insure_trap_error`. This is a good place to insert a breakpoint if you are working with a debugger.

The following functions show the current status of memory and can be called either from your program or the debugger. Remember to add prototypes for the functions you use, particularly if you are calling these C functions from C++ code.

1. `int _Insure_mem_info(char *pmem);`

   Displays information that is known about the block of memory at address `pmem`. (Returns zero.)

2. `int _Insure_ptr_info(char **pptr);`

   Displays information about the pointer at the indicated address. (Returns zero.)

The following function lists all currently allocated memory blocks, including the line number at which they were allocated. It can be called directly from your program or from the debugger.

```
long _Insure_list_allocated_memory(int mode);
```

The `mode` can be chosen from any of the following options:

- `0` - Just the total allocation
- `1` - “Newly-Allocated” or reallocated blocks
- `2` - Everything

Sample Debugging Session

The use of these functions is best illustrated by example. Consider the following program:

```
/*
 * File: bugsfunc.c
 */
#include <stdlib.h>

main()
{
    char *p, *q;

    p = (char *)malloc(100);

    q = "testing";
    while(*q) *p++ = *q++;

    free(p);
    return (0);
}
```

Compile this code under Insure++ in the normal manner (with the `-g` option), and start the debugger in the normal manner.

Note: The instructions shown here assume that the debugger you are using is similar to `gdb`. If you are using another debugger, similar commands should be available.

```
$ gdb bugsfunc
GNU gdb 5.1
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show war-
ranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) break main
Breakpoint 1 at 0x80499e6: file bugsfunc.c, line 7.

(gdb) run
Starting program: /home/Insure++/examples/c/bugsfunc
Breakpoint 1, main (_Insight_argc=1,
_Insight_argv=0xbffff004) at bugsfunc.c:7
7       {
```

If the debugger has trouble recognizing and reading the source file, you may need to use the `rename_files on` option. See “Configuration Options” on page 162 for more information about this option.

It is generally useful to put a breakpoint in `_Insure_trap_error` so that you can get control of the program whenever an error occurs. In this case, we run the program to the error location with the following result

```
(gdb) break _Insure_trap_error
Breakpoint 2 at 0x40143017: file UserInterface.cc, line 303.
```

Note: The above may not work if you have linked against the shared Insure++ libraries (the default). If you cannot set a breakpoint as shown above, it is because the shared libraries are not loaded by the debugger until the program begins to run. You can avoid this problem by setting a breakpoint on `main` and running the program until that breakpoint is hit, then setting the breakpoint on `_Insure_trap_error`.

```
(gdb) c
Continuing.
[bugsfunc.c:15] **FREE_BODY**
>>     free(p);

  Freeing memory block from body: p

  Pointer    : 0x0804b9cf
  In block   : 0x0804b9c8 thru 0x0804ba2b (100 bytes)
                 p, allocated at bugsfunc.c, 10
                           main()  bugsfunc.c, 10

  Stack trace where the error occurred:
                           main()  bugsfunc.c, 15

**Memory corrupted.  Program may crash!!**

Breakpoint 2, _Insure_trap_error () at UserInterface.cc:303
303     }
Current language:  auto; currently c++
(gdb)
```

The program is attempting to free a block of memory by passing a pointer that doesn't indicate the start of an allocated block. The error message shown by Insure++ identifies the location at which the block was allocated and also shows us that the variable p has been changed to point into the middle of the block, but it doesn't tell us where the value of p changed.

We can use the Insure++ functions from the debugger to help track this down. Since the program is already in the debugger, we can simply add a breakpoint back in `main` and restart it.

```
$ gdb bugsfunc
GNU gdb 5.1
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show war-
ranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) break bugsfunc.c:10
```

```
Breakpoint 1 at 0x80499ed: file bugsfunc.c, line 10.

(gdb) run
Starting program: /home/Insure++/examples/c/bugsfunc
Breakpoint 1, main (_Insight_argc=1,
  _Insight_argv=0xbffff004) at bugsfunc.c:10
10          p = (char *)malloc(100);

(gdb) print _Insure_ptr_info(&p)
  Uninitialized
$1 = void
```

To see what is currently known about the pointers `p` and `q`, we can use the `_Insure_ptr_info` function

Note: The `_Insure_ptr_info` function expects to be passed the *address* of the pointer, not the pointer itself. To see the contents of the memory indicated by the pointers, use the `_Insure_mem_info` function.

```
(dbx) print _Insure_ptr_info(&p)
  Uninitialized
(dbx) print _Insure_ptr_info(&q)
  Uninitialized
```

Both pointers are currently uninitialized, as would be expected.

To see something more interesting, we can continue to line 13 and repeat the previous steps.

```
(gdb) break 13
Breakpoint 2 at 0x8049bef: file bugsfunc.c, line 13.
(gdb) cont
Continuing.

Breakpoint 2, main (_Insight_argc=1,
_Insight_argv=0xbffff004) at bugsfunc.c:13
13          while(*q) *p++ = *q++;
(gdb) print _Insure_ptr_info(&p)
  Pointer : 0x0804b9c8 (heap)
  Offset  : 0 bytes
  In Block: 0x0804b9c8 thru 0x0804ba2b (100 bytes)
            p, allocated at bugsfunc.c, 10
$3 = void
```

The variable `p` now points to a block of allocated memory. You can check on all allocated memory by calling `_Insure_list_allocated_memory`.

```
(gdb) print _Insure_list_allocated_memory(2)
1 allocated memory block, occupying 100 bytes.
[bugsfunc.c:10] 0x0804b9c8-0x0804ba2c (100 bytes).
$4 = 100
```

Finally, we check on the second pointer `q`.

```
(gdb) print _Insure_ptr_info(&q)
  Pointer : 0x0804a2d6 (global)
  Offset  : 0 bytes
  In Block: 0x0804a2d6 thru 0x0804a2dd (8 bytes)
            q, declared at bugsfunc.c, 12
$5 = void
```

Everything seems OK at this point, so we can continue to the point at which the memory is freed and check again.

```
(gdb) break 15
Breakpoint 3 at 0x8049d87: file bugsfunc.c, line 15.
(gdb) c
Continuing.

Breakpoint 3, main (_Insight_argc=1,
_Insight_argv=0xbffff004) at bugsfunc.c:15
15          free(p);
(gdb) print _Insure_ptr_info(&p)
  Pointer : 0x0804b9cf (heap)
  Offset  : 7 bytes
  In Block: 0x0804b9c8 thru 0x0804ba2b (100 bytes)
            p, allocated at bugsfunc.c, 10
$6 = void
```

The critical information here is that the pointer now points to an offset 7 bytes from the beginning of the allocated block. Executing the next statement, `free(p)`, will now cause the previously shown error, since the pointer doesn't point to the beginning of the allocated block anymore.

Since everything was correct at line 12 and is now broken at line 15, it is simple to find the problem in line 13, where pointer p is incremented while looping over q.

Tracing

Tracing is a very useful enhancement of Insure++ for C++ programmers. Because C++ is such a complicated language, programmers may never know which functions are being called or in which order. Some functions are called during initialization before the main program begins execution. Tracing provides the programmer with the ability to see how functions, constructors, destructors, and more are called as the program runs.

Insure++ prints a message at the entry to every function which includes the function name, filename, and line number of the command that called it.

A typical line of output from tracing looks like this:

```
function_name filename, line_number
```

By default, the output is indented to show the proper depth of the trace.

Activating Tracing

By default, tracing is turned off. The easiest way to turn tracing on is to set the `trace on` value. This turns on tracing for the entire program. See "Advanced Configuration Options Used by Insure++" on page 167 for more information about this option.

Note: To get a full trace, you must use the `-g` compiler switch on your `insure` compile line. To get file names and line numbers in the trace output, you must use the `stack_internal on` option when compiling your program. You may not want to always do this, because your program will slow down while every function call prints information.

This problem can be minimized by selectively turning on tracing during the execution of your program only in those sections of the code where you need it most. This can be done using the special Insure++ command

```
        void _Insure_trace_enable(int flag)
```

where `flag` = 0 turns tracing off, and `flag` = 1 turns tracing on.

There is an additional special Insure++ function that works with tracing. This function may be used to add your own messages to the trace

```
void _Insure_trace_annotate(int indent, char *format, ...)
```

where `indent` = 0 means string is placed in column zero, `indent` = 1 means string will be indented at proper level, and `format` should be a normal `printf`-style format string.

Directing Tracing Output To A File

You can direct tracing output to a specific file by setting a `trace_file filename` value in the **Advanced tab> Advanced Configuration Settings for Insure++**. When you use this option, Insure++ prints a message reminding you where the tracing data is being written. If you would like to eliminate these reminders, you can use the `trace_banner off` option.

Example

The following code can be found in the `examples\cpp` directory as the file `trace.cpp`.

```
/*
 * File: trace.C
 */
int twice(int j) {
        return j*2;
}

class Object {
public:
        int i;
        Object() {
                i = 0;
        }
        Object(int j) {
                i = j;
        }
        operator int() { return twice(i); }
};

int main() {
        Object o;
        int i;

        i = o;
        return i;
}
```

If you compile and link `trace.cpp` with the `-Zoi "stack_internal on"` option, and then run the executable with the `trace on` value set, you will see the following output:

```
main                             [called by non-insure code]
  Object::Object                 trace.C, 21
  Object::operator int           trace.C, 24
    twice                        trace.C, 17
```

For more information about these and other options see “Advanced Configuration Options Used by Insure++” on page 167.

Signals

In addition to its other error checks, Insure++ also traps certain signals. It does this by installing handlers when your program starts up. These do not interfere with your program's own use of signals - any code which manipulates signals will simply override the functions installed by Insure++.

Signal Handling Actions

When a signal is detected, Insure++ does the following

- Prints an informative error.
- Logs the signal in the Insure++ report file, if one is being used.
- Calls the function `_Insure_trap_error`.
- Takes the appropriate action for the signal.

If this last step will result in the program terminating, Insure++ attempts to close any open files properly. In particular, the Insure++ report file will be closed. Note that this can only work if the program hasn't crashed the I/O system. If, for example, the program has generated a "bus" or similar error, it might not be possible to close the open files. In the worst of all possible scenarios you will simply generate another (fatal) signal when Insure++ attempts to clean up.

Which Signals Are Trapped?

By default, Insure++ traps all signals. You can subtract from this list by adding lines to your `.psrc` file and re-running the program.

Signals are removed with

```
        insure++.signal_ignore SIGINT SIGQUIT SIGTERM
```

Note: You can omit the `SIG` prefix if you wish.

Working With Inuse

Inuse is a graphical tool designed to help developers avoid memory problems by displaying and animating in real time the memory allocations performed by an application.

By watching your program allocate and free dynamic memory blocks, you gain a better understanding of the memory usage patterns of your algorithms and also an idea of how to optimize their behavior.

Inuse allows you to:

- Look for memory leaks.
- See how much memory your application uses in response to particular user events.
- Check on the overall memory usage of your application to see if it matches your expectations.
- Look for memory fragmentation to see if different allocation strategies might improve performance.

Running the Inuse User Interface

The Inuse Graphical User Interface (GUI) does nothing but wait for programs to start and connect to it, at which point it can display their memory activity. When these programs terminate, the Inuse window remains active until you choose to exit it. This allows you to analyze the data gathered during a program’s run once the program has completed execution.

If you exit Inuse while a program is still running, that program will continue running as usual but will stop sending memory activity data. You will have to start the inuse process again before memory activity can be displayed.

To run the example shown in this section, execute the commands

```
cp /usr/local/insure/examples/c/slowleak* .inuse
```

The first of these commands copies a set of example files to your local working directory, while the second starts Inuse. Normally, you will only have to execute the `inuse` command once. The `inuse` process will remain running in the background, accepting display requests from any application that you choose to run.

Compiling and Linking For Inuse

To use Inuse, you need to link your executable program with the `insure` command. Compile the objects and libraries that make up your application with your regular compiler and link them with Insure++ to create a new executable.

Note: The `insure` command replaces the `insight` command used in previous versions of Inuse.

Compile and link the sample program with the `insure` command:

```
cc -c slowleak.c
insure -o slowleak slowleak.o
```

Note: If you are using a compiler other than `cc`, you can tell Insure++ to use the correct compiler during the link step by adding a line such as `insure++.compiler gcc` to a `.psrc` file.

The first command compiles the source file into an object module, while the second links with the special Inuse dynamic memory library.

Inuse is already available to your program if you are compiling your program with Insure++ to debug your code. With earlier versions, if you compiled with Insure++, Inuse would also display information about the Insure++ runtime as it performs error detection. This is no longer the case, because Insure++ now uses two separate heaps for the program. Nevertheless, we still recommend the method described above.

Enabling Runtime Activity Display

Now that your program is linked with the appropriate libraries, you will need to enable the runtime memory activity display. To do this, you must add the following option to your `.psrc` file.

```
insure++.inuse on
```

Running the Application

Once you have enabled the runtime display, you can run your application just as you would normally. To run the example application `slowleak`, type the command

```
slowleak
```

The Inuse Display

When you start the example application, it will connect to Inuse. The Inuse display shows which applications are currently linked to the GUI. From this screen you can open any of Inuse's visual reports.

For a complete description of the Inuse display, see "Running Inuse" on page 116.

Is There a Bug In the Slowleak Program?

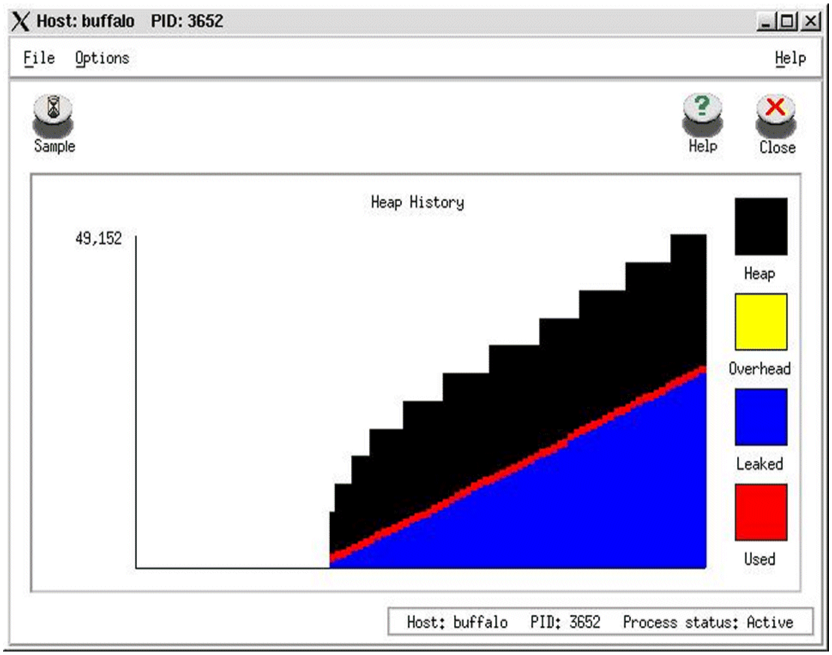
Clicking on the **Hist** button on the Inuse display will open up the **Heap History** window.

Watch the window for a few moments as the `slowleak` program continues to run. The window should soon appear similar to the one shown below in Figure 1.

Figure 1 clearly shows that the program is continuously allocating more and more memory - the classic symptom of a memory leak. This type of pattern in an Inuse report is important to watch for in your own applications, since it probably means that something is wrong.

To find the cause of the problem, you can either look at the source code manually and attempt to figure it out yourself, or simply compile the program and run it with Insure++ using the following commands

```
insure -g -o slowleak slowleak.c
slowleak
```



Note: To use Inuse, you only need to link with the `insure` command. To find the memory leak, you need to compile and link with Insure++, as shown above. For a description of detecting memory leaks with Insure++, see “Memory Leaks” on page 38.

To see the difference in Inuse's output for correct and incorrect programs, you can either fix the problem in the `slowleak` example or copy, compile, and link the corrected version, `noleak`, with the following commands:

```
cp /usr/local/insure/examples/c/noleak.c .
cc -c noleak.c
insure -o noleak noleak.o
noleak
```

If you exited Inuse, you will need to start it again before running the last of the commands shown above. You also need to have the `insure++.inuse on` option set in your `.psrc` file to enable the graphical display, as explained in "Enabling Runtime Activity Display" on page 114. Inuse can be run independently of Insure++, meaning no relinking is necessary!

Running Inuse

The basic steps involved in using Inuse are:

- Running the GUI.
- Linking your program to Inuse.
- Adding the `inuse on` option to your `.psrc` file and running the application program.

During runtime, you can view and manipulate the displays shown by the GUI. You can even watch the memory allocation as you single step through your program from a standard code-oriented debugger. This section will cover each of these steps in detail.

The Basics

You must start the Inuse program before you attempt to display results from any user application. (If you try to run an application before starting Inuse it will run normally, without displaying any memory activity.) In normal use, you should enter the `inuse` command once and simply leave it running as a background process:

```
inuse
```

You should compile your code with your regular compiler and then link with Insure++ as shown below:

```
insure -o foo foo.c
```

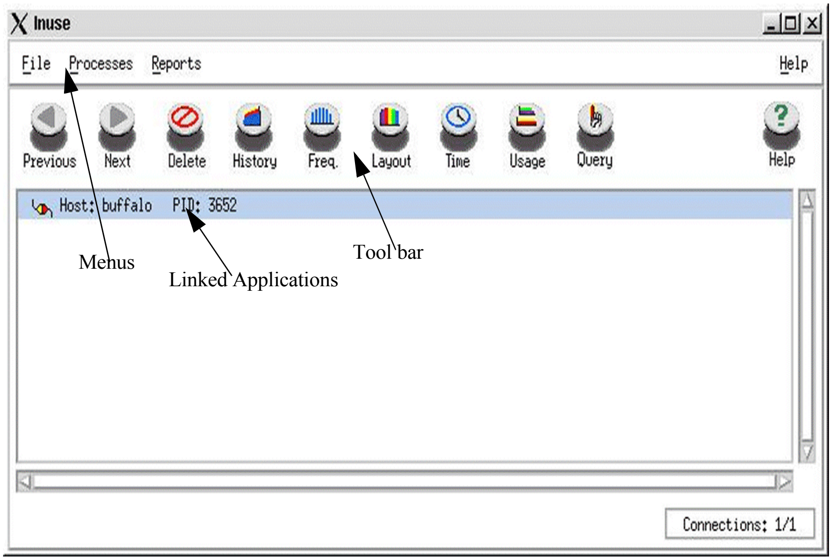
To enable runtime display of memory activity, you need to set the following option in your `.psrc` file:

```
insure++.inuse on
```

Inuse can be linked simultaneously with any number of application programs. By turning this option on and off, you can control when your programs connect to Inuse. If you exit Inuse, you must restart it before running any applications for which you wish to display memory activity.

The Inuse GUI

Executing the `inuse` command opens the Inuse GUI. When you connect a program to Inuse, the connection will appear in the main window. The “plugged in” symbol next to the connection shows that the program is currently sending data to Inuse. If you tell Inuse to stop receiving memory data from the program, this symbol will change to a stop sign. When the program finishes its run or is terminated, the symbol is replaced with a “RIP.”



Note: The “look and feel” of a windowing application will vary quite significantly from system to system. As a result, the version of the window that you see might differ from that shown above.

The Inuse Menu Bar

The menu bar contains commands that manage Inuse memory functions. Available commands are detailed below:

File

- **Load** a file for inspection through Inuse.
- **Save** the currently open file.
- Save a file under a new name and location using **Save as**.
- Save all files currently open in Inuse using **Save all**.
- Save all files currently open in Inuse to a certain location and name using **Save all as**.
- Choose an executable file and **Run** it.
- **Exit** the Inuse GUI.

Processes

- Attach a **Label** to a linked application.
- Use **Next** or **Prev** to choose among linked applications.
- **Del** (delete) a program from the list.
- **Stop** receiving memory display information from a linked program.
- **Step** through one allocation or free request from the program at a time.

Reports

The types of reports Inuse generates are described below. For more information on these reports, see “Inuse Reports” on page 92.

- **Heap History**

  This graph displays the amount of memory allocated to the heap and the user process as a function of real (that is, wall clock) time. This display updates periodically to show the current status of the application, and can be used to keep track of the application over the course of its execution.

- **Block Frequency**

  This graph displays a histogram showing the number of blocks of each size that have been allocated. It is useful for selecting potential optimizations in memory allocation strategies.

- **Heap Layout**

  This graph shows the layout of memory in the dynamically allocated blocks, including the free spaces between them. You can use this report to “see” fragmentation and memory leaks.

  You can scan through different areas of the layout by pressing the Fast Left (**F.left**), Fast Right (**F.right**) and **left** and **right** buttons on the Heap Layout tool bar. You can also zoom in (+) and out (-) of the layout by pressing the **zoom** buttons. (These options are also available in the **Controls** menu.)

  Clicking any block in the heap layout will tell you the block’s address, size, and status (free, allocated, overhead, or leaked). Clicking an allocated or leaked block will also open a window telling you the block id, block address, stack size, and stack trace for the selected block.

- **Time Layout**

  This graph shows the sequence of allocated blocks. As each block is allocated it is added to the end of the display. As blocks are freed, they are marked green. From this display, you can see the relative size of blocks allocated over time. For example, this will allow you to determine if you are allocating a huge block at the beginning of the program or many small blocks throughout the run.

- **Usage Summary**

  This bar graph shows how many times each of the memory manipulation calls has been made. It also shows the current size of the heap and the amount of memory actively in use. (The heap fragmentation can be computed simply from these numbers as `(total-in_use)/total`).

- **Usage Comparison**

  Graphically compares memory from different runs of one executable or among runs of different executables.

- **Query**

  The query function enables you to “view” blocks of memory allocated by your program according to their id numbers, their size, and/or their stack traces. You can edit the range of the query according to block id, block size, and stack trace.

  By “grouping” blocks of memory in this way, you can better understand how memory is being used in your program. The range options let you narrow or broaden your query to your specifications. For example, you can see how much memory is being allocated from a single stack trace or by the entire program combined. For each query you can choose whether you receive a detailed (i.e. containing block id, block size, and stack trace information) or summarized report.

Help

Online help feature for Inuse. Options include:

- **Overview**: Provides a general introduction to the Inuse feature.
- **Inuse Reports**. Describes the types of reports Inuse generates.
- **Inuse Queries**. Describes the uses of the query function in Inuse.
- **About Inuse**. Displays the version of Insra you are running, as well as Parasoft contact information.

Tool Bar

The tool bar has icons for most menu options. Clicking an icon selects that option. The following table lists the available tool bar icons.

Previous	Goes to the previous message in the Inuse GUI.
Next	Goes to the next message in the Inuse GUI.
Delete	Deletes a selected message from the Inuse GUI.
History	Opens a history report for a linked application.
Freq	Opens a block frequency report for a linked application.
Layout	Opens a heap layout report for a linked application.

Time	Opens a time layout report for a linked application.
Usage	Opens a usage summary report for a linked application.
Query	Opens the query manager GUI for a linked application.
Help	Accesses the online help menu for the Inuse application.

Block Color In Inuse

An important visual aid in Inuse is the use of colors to represent the various properties of the heap. These colors are listed below:

Color	Description
Black	Indicates the total memory allocated to the heap. This is usually the amount of memory that gets swapped to disk whenever your application is swapped from memory, regardless of whether or not you are actually using it.
Blue	Denotes leaked blocks as reported by Insure++. Only available if running Inuse and Insure++ together.
Green	Free space that is available to be allocated.
Red	Denotes allocated blocks.
Yellow	"Overhead" associated with each block. Normally the system keeps a small amount of memory with each allocated (and maybe free) block for its bookkeeping information. This memory cannot be used by the application, although its impact can be reduced by allocating fewer large blocks rather than many small ones.

Selecting Reports

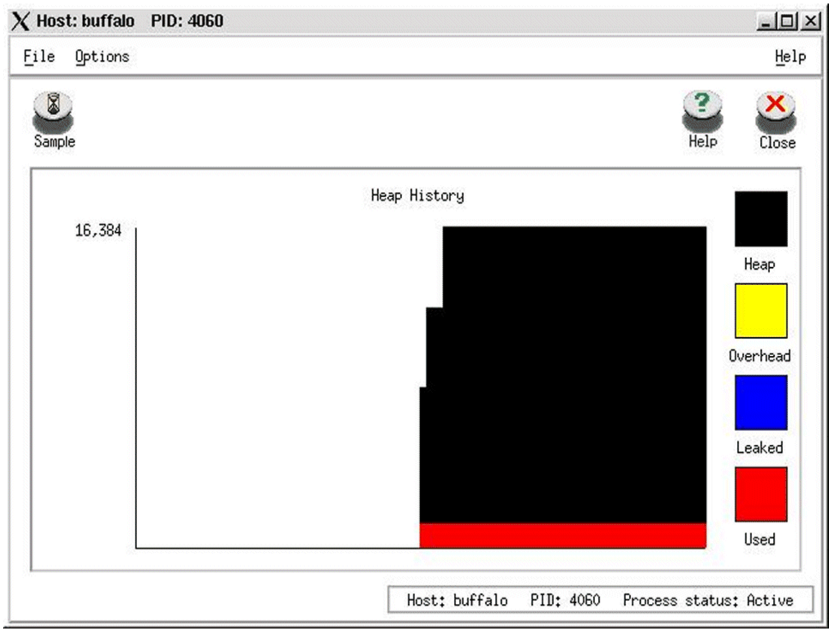
Which report is most useful depends on what you are trying to learn about your application.

- If you want to see how much memory is being used, try the usage summary and heap history reports.
- If you want to optimize your memory allocation strategy, perhaps by building your own allocator for blocks of certain sizes, the block frequency graph is appropriate.
- If you want to study the heap fragmentation caused by your algorithm, and understand the way that memory blocks are laid out, you should use the heap layout graph.
- If you want to see correlations between block id, block size, and stack traces in your program, use the query option.

The following sections contain detailed descriptions of each Inuse report option.

The Heap History Report

Clicking the **History** button or selecting **Heap History** from the Reports menu opens the Heap History window. If your program is running when you open this report, you will be able to monitor the amount of memory allocated by the program as it executes. If the program has ended its run, you can see how much memory was allocated across the history of the run.



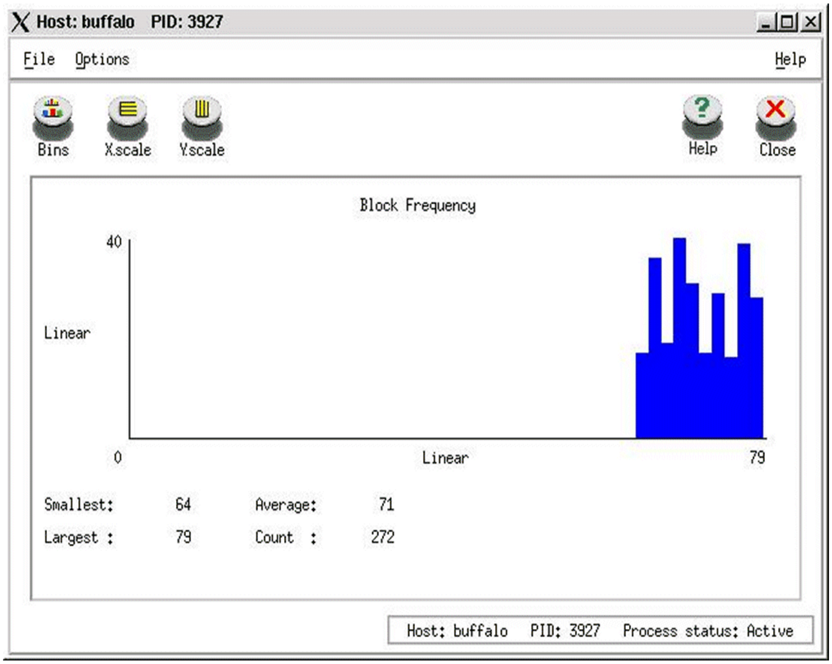
Host: buffalo PID: 4060 Process status: Active

The black area indicates the total size of the heap. The red, yellow, and blue areas constitute the make-up of the heap. The red area represents the amount of memory allocated by the program. The yellow area represents the amount of “overhead” associated with the memory (but which cannot be used by the application). The blue area represents the amount of leaked memory.

To change the sampling rate, that is, how much time passes before the graph is updated, press the **Sample** button or select **Sampling** from the Options menu.

The Block Frequency Report

The block frequency report shows what sizes of blocks are typically being allocated by your program. If you are allocating many small blocks, you may want to switch to a different memory allocation scheme which groups many small allocations into several larger ones. A “block frequency” graph might look like the one below.



Smallest: 64 Average: 71
Largest : 79 Count : 272

Host: buffalo PID: 3927 Process status: Active

The Block Frequency graph groups together blocks that are similarly sized. Each bin (column) in the graph represents the number of blocks in a particular size range. Clicking a bin will show you the number and size range of blocks contained in that bin.

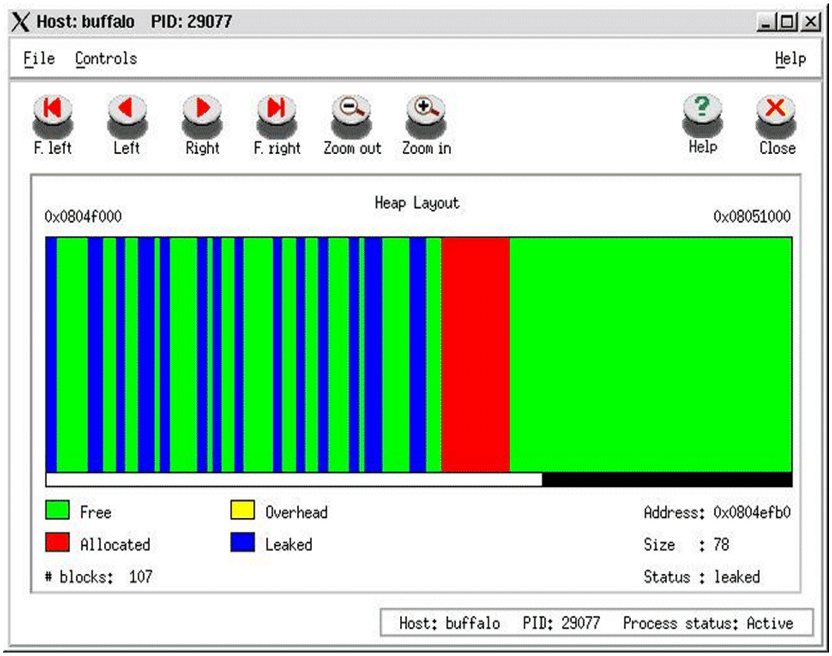
The right-most bin includes all blocks above a certain size. If this bin is very high, click to see the size range covered by that bin. If the range is fairly wide, you can rescale the graph to include more bins by pressing the **Bins** button or selecting **Bins** from the Options menu. Entering a number *higher* than the number currently shown should *narrow* the size range for each bin.

Likewise, entering a number that is *lower* than the number currently shown should *increase* the size range included in each bin. If all block sizes are currently represented on the graph, increasing the bin number will have no effect.

You can alternate between a linear or logarithmic scale by pressing the **Xscale** and **Yscale** buttons. Pressing the **Xscale** button will toggle the x-axis between linear and logarithmic scales. Pressing the **Yscale** button will toggle the y-axis between linear and logarithmic scales. Selecting **Horiz.log/log** or **Vert.log/log** from the Options menu will also toggle the x- or y-axis.

The Heap Layout Report

The heap layout report shows the status of blocks in the program's heap. Blocks are either free, allocated, overhead, or leaked. Click a block to see its address, size, and status. This information is shown in the lower right corner of the display screen.



Host: buffalo PID: 29077

Heap Layout

0x0804f000 0x08051000

🟩 Free 🟨 Overhead

🟥 Allocated 🟦 Leaked

blocks: 107

Address: 0x0804efb0
Size : 78
Status : leaked

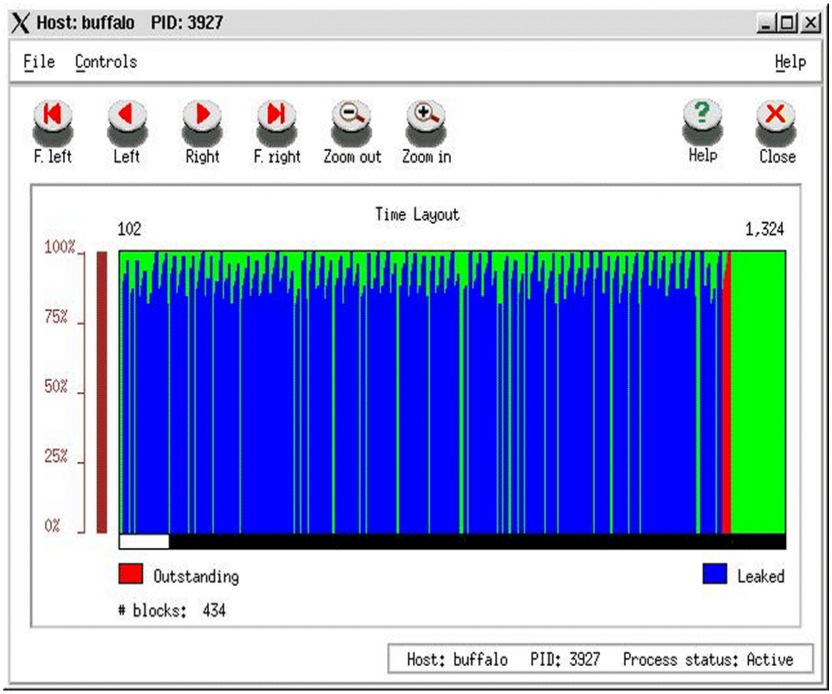
Host: buffalo PID: 29077 Process status: Active

If the block is allocated or leaked, a Memory Information window will also appear. The memory information window contains the block's id, address, stack depth, and stack trace.

You can scan through different areas of the heap layout by pressing the Fast Left (**F.left**), Fast Right (**F.right**) and **left** and **right** buttons on the Heap Layout tool bar. You can also zoom in (+) and out (-) of the layout by pressing the **zoom** buttons. (These options are also available in the **Controls** menu.)

The Time Layout Report

The time layout report shows how memory blocks are allocated across the run of the program. As each block is allocated, it is added to the end of the display. As blocks are freed, they are marked green.



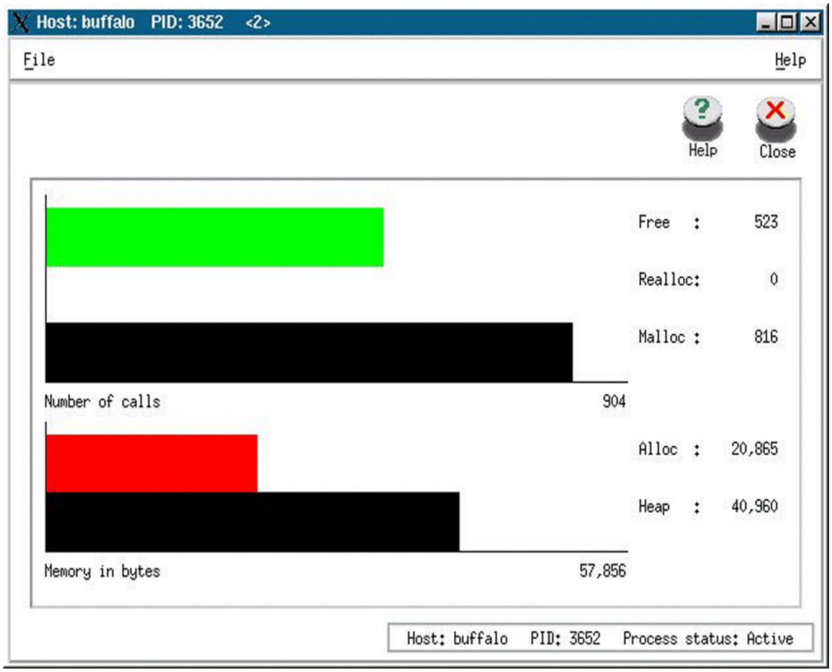
From this display, you can watch how memory is allocated over the run of your program. As you see the patterns in which memory blocks are allocated and freed over time, you can better optimize your program's use of memory. Memory leaks are also shown on this display.

You can scan through the run by pressing the Fast Left (**F.left**), Fast Right (**F.right**) and **left** and **right** buttons on the Time Layout tool bar. You can also zoom in (+) and out (-) of the layout by pressing the **zoom** buttons. (These options are also available in the **Controls** menu.)

The Usage Summary Report

The usage summary report shows how much memory you are using and how often calls have been made to each category of memory allocation functions `malloc`, `realloc`, and `free`. Note that the `malloc` category includes all functions which allocate dynamic memory, for example `calloc`, `memalign`, and `XtMalloc`.

Similarly, the `realloc` category includes all functions which relocate or resize dynamic memory blocks. The `free` category includes all functions which free dynamic memory. You can calculate the number of blocks currently allocated by subtracting the number of `free`s from the number of `malloc`'s. A typical "usage summary" graph might look like the following:



Host: buffalo PID: 3652 Process status: Active

The number given by "Alloc" is the total number of bytes allocated dynamically by your program. The number given by “Heap” is the total number of bytes currently allocated by the system to your program’s heap.

The number given with “Number of calls” and “Memory in bytes” is simply the extreme value on the x-axis for each graph. The limit of each graph will change as Inuse updates the display with more memory allocation function calls.

Query Reports

Query is a powerful tool that makes it easier for you to understand how memory is being used by your program. With Query, you can find out exactly how much memory is being allocated to blocks of a particular size or location; how much memory is being allocated from a particular path; find out the stack traces of blocks of a particular size or location; and much, much more.

By creating queries, you will be creating a “model” of your program’s memory use. You will be able to “look” at it from different angles and approaches, learning more and more with each successive report. As you come to understand how and where your program uses memory, you will be able to better optimize your program’s memory use.

For example, if the Block Frequency report shows that your program is using many small allocations (creating a lot of memory overhead), you might want to know exactly how much memory is being allocated to these small blocks. You will also want to know which part of the code is responsible for creating them.

To find out how your program is distributing memory across block sizes, you can run a query that shows how much memory is being allocated to each size block.

If you find that your small blocks contain a lot of memory, you can then run a query that gives you the block ids and stack traces responsible for creating these small blocks.

Armed with this information, you will be able to make simple adjustments to your code that will result in a more effective use of memory.

You can run queries on any combination of block id, block size, and stack trace. These queries can be as flexible or as restrictive as you choose, as you set the parameters. For more information, see “Editing a Query” on page 134.

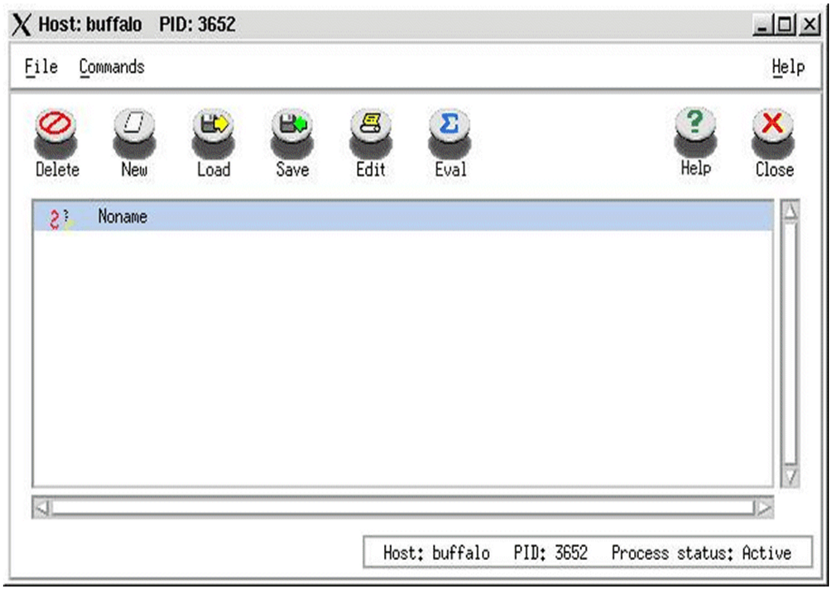
Running queries does not affect your program or its operation. By running different queries and trying different approaches, you will soon see how valuable and informative these reports can be.

Running a Query

Clicking **Query** opens the Query Manager screen. From here you can:

- Press **Delete** to delete the current query.
- Press **New** to start a new query.
- Press **Load** to open a saved query.
- Press **Save** to save the current query for later use.
- Press **Edit** to edit the currently selected query.
- Press **Eval** to evaluate the current query.

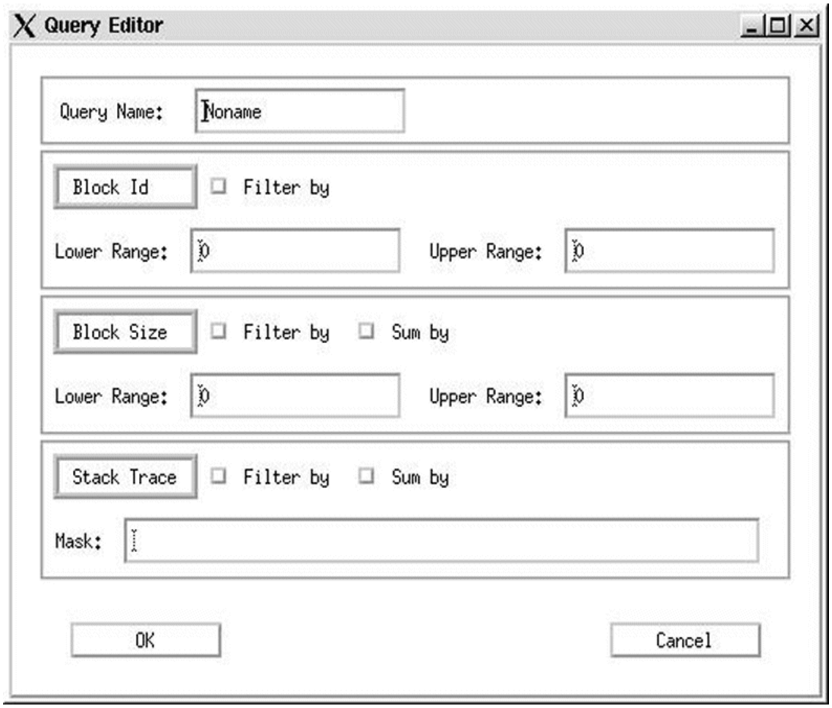
When you open a query, its name will appear in the Query Manager window. If you pressed **New**, the query will be named `NoName`.



Editing a Query

Pressing the **Edit** button on the Query Manager screen will let you edit the currently selected query. The Query Editor window allows you to:

- Change the name of a query.
- Set the lower and upper ranges for the block ids you want to isolate.
- Set the lower and upper ranges for the block sizes you want to isolate.
- Enter an expression to isolate certain stack traces.
- Choose whether to receive a complete report (including block id, block size, and stack trace data) or just a summary of block size and/or stack trace information.



Query Editor

Query Name: Noname

Block Id □ Filter by

Lower Range: 0 Upper Range: 0

Block Size □ Filter by □ Sum by

Lower Range: 0 Upper Range: 0

Stack Trace □ Filter by □ Sum by

Mask:

The default values for a query are `0` for the lower and upper block id and block size ranges and no expressions in the stack trace area. This will produce the widest query, listing all memory blocks allocated by your program.

You can filter a query to isolate particular block ids, block sizes, and/or stack traces. Just enter the values you want to filter for and “check” the **Filter by** box.

The **Sum by** option provides a useful summary of block size and stack trace data. Use it in combination with the **Filter by** option or on its own to get a breakdown of blocks by size and/or stack trace.

A Query Example

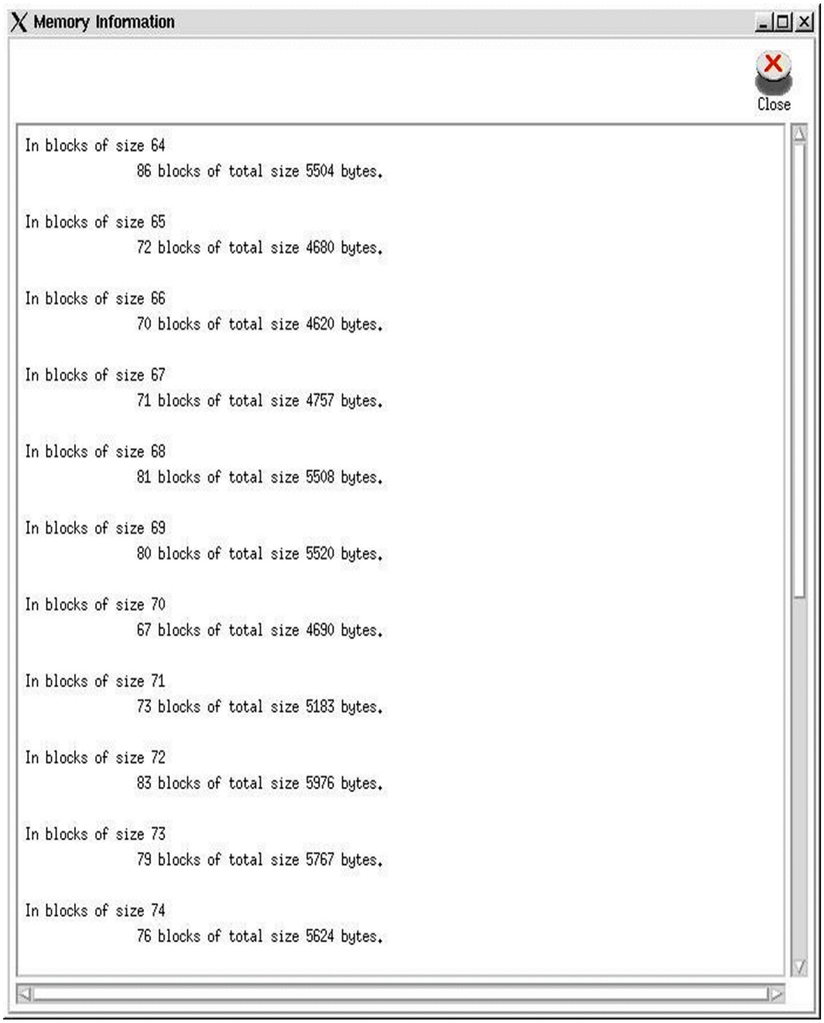
Assume you want to learn more about blocks that are between 60 and 70 bytes in size. In the Block Size area, enter 60 in the lower range and 70 in the upper range. Then click the Block Size **Filter by** box and press **OK**. Running this query will return block ids and stack traces for all blocks allocated by your program that are sized between 60 and 70 bytes.

Checking the Block Size **Sum by** box for this query will return a list showing how much memory (in bytes) is being allocated to each block size in that range.

You can enter filters for any or all three areas in a single query. This can help you find out the exact stack trace(s) responsible for the largest block allocations by your program, or conversely, to find out the size and location of memory allocated from a particular stack trace path.

Once you have edited your query, pressing **OK** returns you to the Query Manager.

To run the query, press the **Eval** button or select **Evaluate Query** from the Commands menu. Query results will appear in a text window, as shown below:



```
In blocks of size 64
                86 blocks of total size 5504 bytes.

In blocks of size 65
                72 blocks of total size 4680 bytes.

In blocks of size 66
                70 blocks of total size 4620 bytes.

In blocks of size 67
                71 blocks of total size 4757 bytes.

In blocks of size 68
                81 blocks of total size 5508 bytes.

In blocks of size 69
                80 blocks of total size 5520 bytes.

In blocks of size 70
                67 blocks of total size 4690 bytes.

In blocks of size 71
                73 blocks of total size 5183 bytes.

In blocks of size 72
                83 blocks of total size 5976 bytes.

In blocks of size 73
                79 blocks of total size 5767 bytes.

In blocks of size 74
                76 blocks of total size 5624 bytes.
```

Working With TCA

TCA (Total Coverage Analysis) lets you get “beneath the hood” of your program to see which parts of your program are actually tested and how often each block is executed. In conjunction with a runtime error detection tool like Insure++ and a comprehensive test suite, this can dramatically improve the efficiency of your testing and guarantee faster delivery of more reliable programs.

Coverage Analysis

The idea behind test coverage is to analyze how many of an application's files, functions, and statements have been executed. This data can be used during development, and particularly during testing, to give some idea of the overall quality of the testing.

The hope is that having information about which parts of an application haven't been tested will enable you to modify or enhance your existing testing procedures to cover the untested portions.

Unfortunately, this is a vain hope with conventional testing methods, because even if the effort is made to achieve 100% test coverage, the tests will usually be heavily biased towards the “important” parts of the code which get executed most often.

Because of this, many organizations do not use test coverage analysis to any great extent in their development process.

The Significance of Runtime Testing

An important difference emerges, however, when one considers the additional power of runtime debugging tools such as Insure++.

The quality of the checking performed by Insure++ is independent of the amount of effort put in by the development or quality assurance team--it performs complete and thorough checking on any piece of code that it executes (and also, of course, a significant amount of compile time checking on code that it merely compiles). As a result, it makes a lot of sense to aim for 100% test coverage if you are using a product such as Insure++, because the testing is much more thorough.

For this reason, Insure++ contains the Total Coverage Analysis (TCA) module, which is a component of the Total Quality Software package designed to guide you to 100% execution of your application during the testing and quality assurance processes.

Using TCA

The coverage analysis system works in a similar way to Insure++. Your application is processed with the insure command instead of your normal compiler. This process first builds a temporary file containing your original source code modified to include checks for coverage analysis, and then passes it to your normal compiler. While your application runs, these modifications cause your application to create a database containing information about which blocks were executed. This database can then be analyzed with a special application (`tca`) to create reports.

Preparing your Code For Coverage Analysis

Since coverage analysis is such a powerful tool when used in conjunction with Insure++, it is automatically enabled whenever you compile an application with the `insure` command, unless you specifically disable it, as described in the section “Options Used By TCA” on page 190. If you only want to perform coverage analysis (i.e. you don't want the normal Insure++ runtime checking), you can compile and link with the `-Zcov` switch.

Note: Please note that if you compile with `insure` and the `-Zcov` option, you must also link with the `-Zcov` option, and vice versa. You cannot use `-Zcov` in only one stage of the build.

An Example - Sorting Strings

To see this process in action, consider the code shown below, which is a modified version of a bubble sorting algorithm.

```
/*
 * File: strsort.c
 */
#include <stdio.h>
#include <string.h>
```

```
bubble_sort(a, n, dir)
    char **a;
    int n, dir;
{
    int i, j;

    for(i=0; i<n; i++) {
        for(j=1; j<n-i; j++) {
            if(dir * strcmp(a[j-1], a[j]) > 0) {
                char *temp;

                temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}

main(argc, argv)
int argc;
char **argv;
{
    int i, dir, length, start;

    if (argc > 1 && argv[1][0] == '-') {
        if (argv[1][1] == 'a') {
            dir = 1; length = argc-2; start = 2;
        } else if (argv[1][1] == 'd') {
            dir = -1; length = argc-2; start = 2;
        }
    } else {
        dir = 1; length = argc; start = 1;
    }
    bubble_sort(argv+start, length, dir);
    for (i = 2; i < argc; i++)
        printf("%s ", argv[i]);
    printf("\n");
    return 0;
}
```

This program sorts a set of strings supplied as command line arguments in either ascending or descending order, according to the settings of the command line switches.

```
strsort -a s1 s2 s3 ...
```

Sorts strings in ascending order.

```
strsort -d s1 s2 s3 ...
```

Sorts strings in descending order.

```
strsort s1 s2 s3 ...
```

Sorts strings in ascending order.

If you wish to try the commands shown in this section, you can use the source code supplied with the Insure++ examples by executing the command

```
        cp /usr/local/insure/examples/c/strsort.c .
```

To compile and execute this program with both runtime error detection and coverage analysis enabled, simply use the normal `insure` command

```
        insure -g -o strsort strsort.c
```

In addition to compiling the program, this will create a file called tca.map which describes the layout of your source file. We can now perform a set of tests on the application. A few samples are shown below. The statements beginning with the `$ symbol` are the commands executed and the remaining text is the response of the system.

```
$ strsort -a aaaa bbbb
aaaa bbbb
** TCA log data will be merged with tca.log **

$ strsort -a bbb aaa
aaa bbb
** TCA log data will be merged with tca.log **

$ strsort -d aaa bbbb
```

```
bbbb aaa
** TCA log data will be merged with tca.log **

$ strsort -d bbb aaa
bbb aaa
** TCA log data will be merged with tca.log **
```

Note the following features from this example:

- Each time the application is executed, the coverage analysis module issues a message indicating that information is being added to a `log` file, which is created the first time the program is run. This file contains the coverage information for one or more test runs.
- Insure++ issues no error messages during any execution of the program.

Analyzing Test Coverage Data

Analysis of the test coverage data is performed using the `tca` command. There are several summary levels. The ones we will introduce here are:

- Overall summary (`default`) - Shows the percentage coverage at the application level - i.e., summed over all program entities.
- Function summary (`-df` switch) - Displays the coverage of each individual function in the application.
- Source code summary (`-ds` switch) - Displays the source code with each block marked as executed (`.`) or not (`!`).

To see these commands in action, execute the command

```
tca tca.log
```

This displays the top level summary, shown below.

```
COVERAGE SUMMARY
================
     1 block  untested
    12 blocks tested

92% covered
```

As can be seen, the overall coverage is quite high. However, one program block remains untested. To find out which one is untested, execute the command

```
         tca -df tca.log
```

This command displays the function level summary, and includes functions that are 100% tested.

```
COVERAGE SUMMARY - by function
================

   blocks    blocks    %cov =      functions
 untested    tested    %tested
 -------------------------------------------------

    0       4     100%    bubble_sort  [strsort.c, line 7-13]
    1        8     88%     main    [strsort.c, line 26-45]
```

From this listing, you can see that the function `bubble_sort` is completely tested, but that one block in `main` remains untested. To find out which one, execute the command

```
         tca -ds tca.log
```

This results in the following output.

```
UNTESTED BLOCKS - by file
===============

FILE strsort.c 92% covered: 1 untested / 12 tested

        /*
         * File: strsort.c
         */
        #include <stdio.h>
        #include <string.h>

        bubble_sort(a, n, dir)
            char **a;
            int n, dir;
        {
            int i, j;
   . ->     for(i=0; i<n; i++) {
   . ->         for(j=1; j<n-i; j++) {
   . ->             if(dir * strcmp(a[j-1], a[j]) > 0) {
                        char *temp;

   . ->                 temp = a[j-1];
                        a[j-1] = a[j];
                        a[j] = temp;
                    }
                }
            }
        }

        main(argc, argv)
        int argc;
        char **argv;
        {
            int i, dir, length, start;

   . ->     if (argc > 1 && argv[1][0] == '-') {
   . ->         if (argv[1][1] == 'a') {
   . ->             dir = 1; length = argc-2; start = 2;
   . ->         } else if (argv[1][1] == 'd') {
   . ->             dir = -1; length = argc-2; start = 2;
                }
            } else {
   ! ->         dir = 1; length = argc; start = 1;
            }
   . ->     bubble_sort(argv+start, length, dir);
            for (i = 2; i < argc; i++)
   . ->         printf("%s ", argv[i]);
   . ->     printf("\n");
            return 0;
        }
```

This listing shows exactly which statements have been executed and which have not. The results show that the untested block corresponds to the case where `strsort` is executed with neither the `-a` nor `-d` command line switches.

Achieving 100% Test Coverage

The previous analysis tells us that we can achieve 100% test coverage in this example by executing the `strsort` program without either of its two switches.To complete testing, execute the program with the command

```
strsort aaa bbbb
```

This produces the following output

```
[strsort.c:15] **READ_NULL**
>>             if(dir * strcmp(a[j-1], a[j]) > 0) {

  Reading null pointer: <argument 2>

  Stack trace where the error occurred:
                          strcmp()  (interface)
                     bubble_sort()  strsort.c, 15
                            main()  strsort.c, 41

**Memory corrupted.  Program may crash!!**

**Insure trapped signal: 11**

  Stack trace where the error occurred:
                          strcmp()  ../sysdeps/generic/
strcmp.c, 39
                          strcmp()  (interface)
                     bubble_sort()  strsort.c, 15
                            main()  strsort.c, 41
Segmentation violation
Abort (core dumped)
Exit 134
```

which indicates that Insure++ has found an error in this code block. (Finding and fixing this error is left as an exercise for you. Remember that if you built the program with the `-Zcov` option, this bug would not have been detected by Insure++).

However, when the command `tca tca.log` is executed, the following output is produced

```
COVERAGE SUMMARY
================
     0   blocks untested
    13   blocks tested

100% covered
```

which indicates that the application has now been 100% tested.

This means that the set of test cases that have been run, including this last one, completely exercised all the code in this application. It makes sense to incorporate these test cases (in conjunction with Insure++) into a quality assurance test suite for this code.

There are several `.psrc` options you can use to control coverage analysis. These are documented in “Configuration Options” on page 162.

How to Use Coverage Analysis

If you are using Insure++, coverage analysis information will be automatically built into your program. At any time after you have run your code you can use the `tca` command to find any blocks which have not been executed. For clarity, the process is broken down into three steps.

- Compiling applications and building their coverage analysis database (usually named `tca.map`).
- Running test cases against applications that have been compiled with coverage analysis enabled, which creates entries in the TCA log file (usually named `tca.log`).
- Running the TCA analysis tool to generate coverage analysis reports from the given coverage log file(s).

You can make your program displays a coverage analysis report when it exits by adding the

```
insure++.summarize coverage
```

option to your `.psrc` file. The `coverage_switches` option lets you set flags to control the output just as though you were passing those switches to `tca`.

Step 1: Compile Time

At compile time, Insure++ creates a database of each file processed, describing how many blocks and statements are in each file and function. This database is called a map file, because it provides TCA with a map of how your program is laid out. By default, the name of this file is `tca.map`, but you can change the name of this file by adding a `coverage_map_file` value to your `.psrc` file.

Ideally, all the files in your application should store their information in the same map file. If your source code is spread across several directories, you will probably want to set the map filename using a full path. For example:

```
insure++.coverage_map_file ~/project.map
```

If you compile several files simultaneously and they are all trying to modify the same map file, you may end up with a corrupt map file. In this case, you will need to delete the original map file and recompile the application you are interested in.

As mentioned before, if you are only interested in coverage information and not debugging, you can add the `-Zcov` option to the `insure` command lines that build your program. Remember to use `-Zcov` consistently, i.e. at both compilation and linking, if you use it at all.

Step 2: Runtime

At runtime, your program (compiled with Insure++) writes a log file, which records the blocks that were actually executed during a specific run. By default, this file is called `tca.log`, but as with the map file, you can change the name of this file by adding a `coverage_log_file` value to your `.psrc` file. Normally, each time you run your program the new log information will be combined with any found in the existing log file, unless the data is not compatible (because you changed your code and recompiled, for example).

Another useful option is to generate a new log file each time your application runs. You can do this by taking advantage of the `%n filename` option, for example

```
insure++.coverage_log_file tca.log.%n
```

In this example, each run would make a new file, such as `tca.log.0`, `tca.log.1`, and so forth. If your program forks, you will need to use this option so that each child creates its own log file.

Step 3: Using TCA to Display Information

After you have created one or more log files, you can use the `tca` command to get the information in which you are interested. TCA normally sends its reports to `stdout`. If you would like to use the graphical version to generate coverage reports, see “The TCA Display” on page 118 for more information. You can specify any number of log files on the command line, and TCA will combine the data before displaying the results. If the log files are not compatible — for example, because they are from different applications — TCA will throw out the ones that do not match the first log file.

TCA will also need to read the map file created at compilation time. Since this name is stored in the log file, you won't normally need to specify it.

By default, TCA will give you a quick summary of how much of your code was tested. Using different options, you can get detailed reports of coverage by `file`, `function`, or even `block`. For each block, TCA can tell you how many times it was executed, summing over all the log files (unless `coverage_boolean` was on at compile time, the default setting).

Note: If a single statement spans several lines of source code, TCA treats the statement as lying on the last line; this is only important for understanding the output of TCA, and does not effect how coverage statistics are calculated.

How Are Blocks Calculated?

Unlike some other coverage analysis tools which work on a line-by-line basis, TCA is able to group your code into logical blocks. A block is a group of statements that must always be executed as a group. For example, the following code has three statements, but only one block.

```
i = 3;
j = i+3;
return i+j;
```

Some of the advantages of using blocks over lines are,

- Lines of code which have several blocks are treated separately.
- Grouping equivalent statements into a single block reduces the amount of data you need to analyze.
- By treating labels as a separate group, you can detect which paths have been executed in addition to which statements.

Note: Conditional expressions containing `&&` or `||` are grouped with the statement they are part of. Also, the three elements of a for loop are treated as part of the `for` statement (for example, `e1`, `e2`, and `e3` in the code fragment `for(e1;e2;e3)`).

The following simple test program shows how the blocks are determined. In this particular example, there are 16 blocks.

```
/*
 * File: coverage.c
 */

#include <ctype.h>

main(int argc, char **argv) {
     int flag;

     if (argc < 2 || !isdigit(argv[1][0])) {
        printf(“Bad argument(s)\n”);
        exit(1);
     }
     switch(atoi(argv[1])) {
     case 1: case 2: case 3:
        flag = 1;
        break;
     case 4:
     case 5:
        flag = 2;
        break;
     default:
        flag = 0;
        break;
     }
     if (flag > 0) flag = 1; else flag = 0;
     printf("Flag is %\n", flag ? "1" : "0");
     exit(0);
}
```

To achieve coverage, run `insure -g -o coverage coverage.c -Zoi "coverage_boolean off"` in the command line. The next code sample shows the output of `tca -ct -ds tca.log` after several test runs with different values (`coverage ; coverage 2 ; coverage 2 ; coverage 4 ; coverage 7 ; coverage 3`).

By looking at this output, you can see which paths have been executed and which have not. Notice that counts are only given at the *beginning* of each block, and not for each statement within each block.

```
BLOCK USAGE - by file
===========

FILE coverage.c 87% covered: 2 untested / 14 tested

        #include <ctype.h>

        main (int argc, char **argv) {
            int flag;
   6 ->     if (argc < 2 || !isdigit(argv[1][0])) {
   1 ->         printf("Bad argument(s)\n");
                exit(1);
            }
   5 ->     switch(atoi(argv[1])) {
            case 1: case 2: case 3:
0/2/1/3 ->         flag = 1;
                break;
            case 4:
            case 5:
```

```
1/0/1 ->         flag = 2;
                break;
            default:
 1/1 ->         flag = 0;
                break;
            }
5/4/1 ->     if (flag > 0) flag = 1; else flag = 0;
   5 ->     printf("Flag is %d\n:", flag ? "1" : "0");
            exit(0);
        }
```

Finally, the next code sample shows the terser output of `tca -ds tca.log`. In this instance, only blocks which have not been executed are marked (corresponding to blocks with a count of zero in the previous figure). The “`!`” character symbolizes not executed. For lines with multiple blocks, you will also see the “`.`“ character which means that group was executed. This is so you can easily identify which blocks on that line were not tested.

Once again, only the first line of code within each block will be marked in this fashion. If you are using the graphical TCA, the first line of the block will be colored as executed (red) while the rest of the block will be colored as not executed (black).

```
UNTESTED BLOCKS - by file
===============

FILE foo.c 87% covered: 2 untested / 14 tested

        #include <ctype.h>

        main (int argc, char **argv) {
            int flag;
   . ->     if (argc < 2 || !isdigit(argv[1][0])) {
   . ->         printf("Bad argument(s)\n");
                exit(1);
            }
   . ->     switch(atoi(argv[1])) {
            case 1: case 2: case 3:
!... ->         flag = 1;
                break;
            case 4:
            case 5:
```

```
 .!. ->         flag = 2;
                break;
            default:
  .. ->         flag = 0;
                break;
            }
 ... ->     if (flag > 0) flag = 1; else flag = 0;
   . ->     printf("Flag is %d\n:", flag ? "1" : "0");
            exit(0);
        }
```

The TCA Display

The TCA display is a graphical representation of the reports generated during runtime. By utilizing this tool, you will be able to view your `tca.log` files with ease. Much like Insra, TCA allows you to load and save files, browse through source code, and even access online help.

The TCA Display may be invoked by calling `tca` with the `-X` switch along with any other command line options.

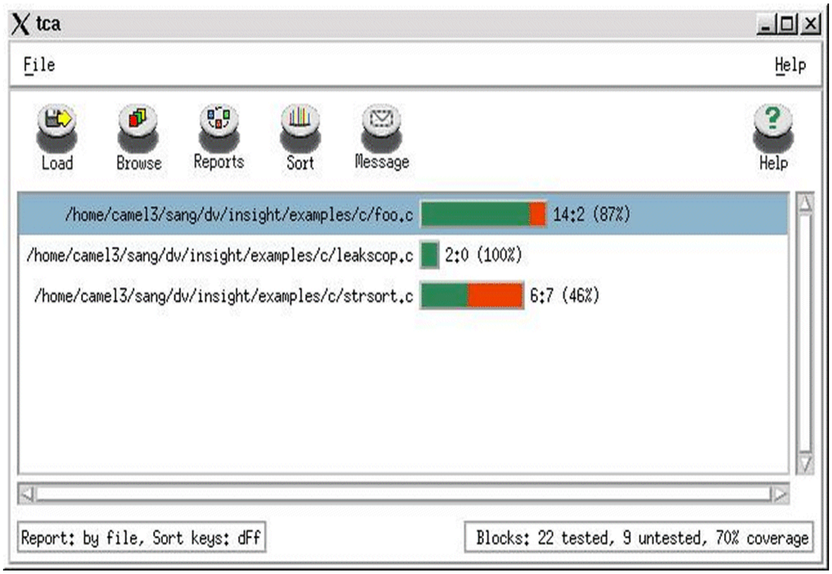
The following subset of TCA command line options are meaningful for the graphical tool, while the remaining unsupported ones are silently ignored.

Command	**Explanation**
`-df`	Display by function
`-do`	Display be object/class
`-dF`	Display by file
`-dd`	Display by directory
`-ns`	Simple function names
`-ne`	Extended function name - include argument types
`-nm`	Include modifiers const/volatile in function arguments
`-ff` name	Only show coverage related to function “name.” This option only applies when `-df` or `-dF` is also specified.

Command	Explanation
`-fo` name	Only show coverage related to object “name.” This option applies when `-do` is also specified.
`-fF` name	Only show coverage related to file “name.” “name” must include the full path to the file. This option only applies when `-dF` or `-df` is also specified.
`-fd` name	Only show coverage related to directory “name.” This option only applies when `-dd` is also specified.
`-s` {keys}	Sort output by keys (`d`, `F`, `n`, `%`, `#`, `b`, 1)
`-ct`	Show hit counts in the source browser.

Loading A Report File

By default, TCA displays a report based on the log files that were included in the command line when the program was started. Coverage statistics from additional files may be included in the report by clicking on the **Load** button and selecting a new log file. The data contained in the newly selected log file is combined with the existing data and a new report is generated.



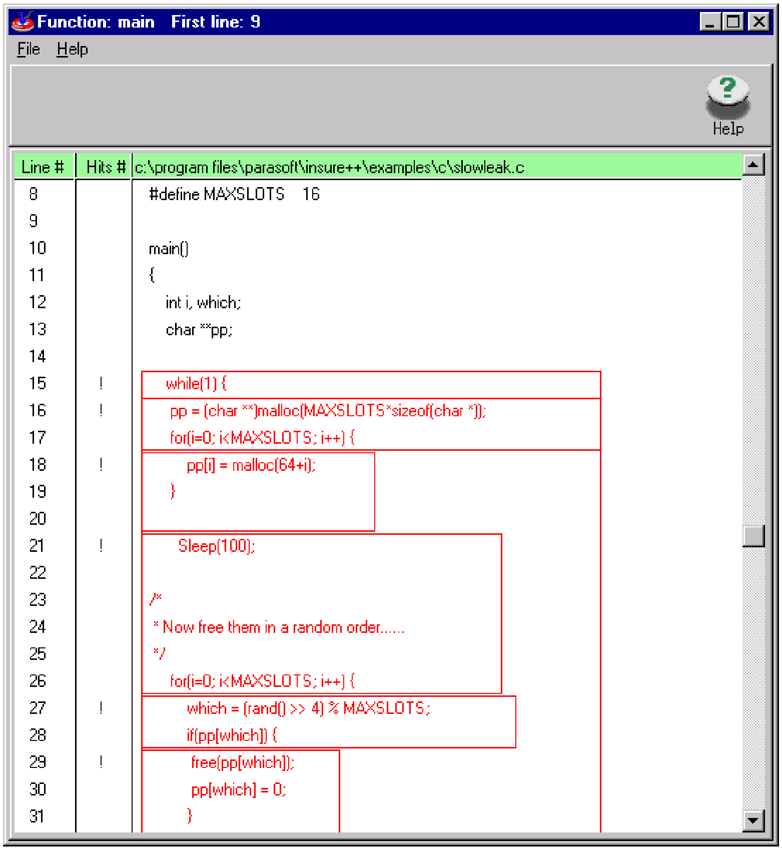
```
/home/camel3/sang/dv/insight/examples/c/foo.c       14:2 (87%)
/home/camel3/sang/dv/insight/examples/c/leakscop.c  2:0 (100%)
/home/camel3/sang/dv/insight/examples/c/strsort.c   6:7 (46%)

Report: by file, Sort keys: dFf        Blocks: 22 tested, 9 untested, 70% coverage
```

Browsing The Source

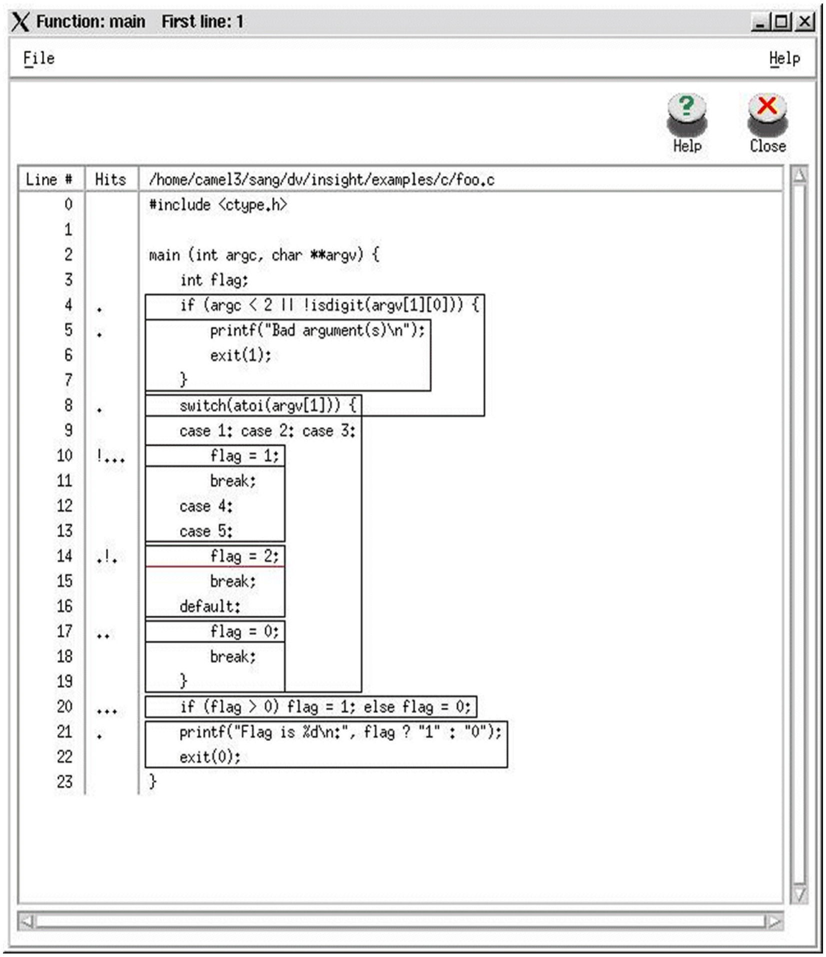
The **Browse** button generates a new window containing the next level of coverage detail. For example, if you are currently displaying a report “by directory,” clicking **Browse** will open a new window displaying a report "by file" for that directory. If you click **Browse** again, you will get another window displaying a report "by function" for the file(s). Clicking **Browse** a final time displays the source code itself, annotated with coverage information for each block.

Double-clicking on a line in the display is the equivalent to selecting a line and clicking the **Browse** button.



```
Line #  Hits #  c:\program files\parasoft\insure++\examples\c\slowleak.c
 8              #define MAXSLOTS    16
 9
10              main()
11              {
12                 int i, which;
13                 char **pp;
14
15        !        while(1) {
16        !         pp = (char **)malloc(MAXSLOTS*sizeof(char *));
17                  for(i=0; i<MAXSLOTS; i++) {
18        !             pp[i] = malloc(64+i);
19                  }
20
21        !           Sleep(100);
22
23              /*
24               * Now free them in a random order......
25               */
26                  for(i=0; i<MAXSLOTS; i++) {
27        !             which = (rand() >> 4) % MAXSLOTS;
28                      if(pp[which]) {
29        !              free(pp[which]);
30                       pp[which] = 0;
31                      }
```

Reports



```
Line #  Hits   /home/camel3/sang/dv/insight/examples/c/foo.c
     0         #include <ctype.h>
     1
     2         main (int argc, char **argv) {
     3             int flag;
     4  .          if (argc < 2 || !isdigit(argv[1][0])) {
     5  .              printf("Bad argument(s)\n");
     6                 exit(1);
     7             }
     8  .          switch(atoi(argv[1])) {
     9             case 1: case 2: case 3:
    10  !...           flag = 1;
    11                 break;
    12             case 4:
    13             case 5:
    14  .!.            flag = 2;
    15                 break;
    16             default:
    17  ..             flag = 0;
    18                 break;
    19             }
    20  ...        if (flag > 0) flag = 1; else flag = 0;
    21  .          printf("Flag is %d\n:", flag ? "1" : "0");
    22             exit(0);
    23         }
```

The level of detail displayed may be changed by clicking on the **Reports** button. A dialog box will appear, allowing you to choose from one of four report types: by directory, by file, by function, or by class.

Sorting

The order in which the coverage information is presented may be modified by clicking on the **Sort** button. A dialog box will appear in which you can enter the sort keys to be used. Any combination of the following keys may be used.

Sort Keys	Explanation
`d`	by directory
`F`	by file name
`f`	by function
`%`	by percent covered
`#`	by number of hits
`b`	by number of blocks

For example, in order to sort by percent covered and decide collisions by the function name, the sort key string should be `%f`. The current sort key string is displayed on the status bar.

Message

This button becomes active when TCA cannot perform a given task. Clicking it opens a window that describes the error(s).

Help

Online help can be accessed in two ways. Context-sensitive help can be accessed by clicking on the **Help** button, which causes a question mark cursor to replace the normal arrow, and then clicking on an area of the GUI. If there is help available for that area or button, a window is displayed with information about how to use the area or button that you clicked. You can also access help from the TCA menu bar.

Setting Preferences

The coverage analysis process consists of the following three stages:

- Compiling applications with Insure++ and building their coverage analysis database (usually named `tca.map`).
- Running test cases against applications that have been compiled with coverage analysis enabled, which creates entries in the TCA log file (usually named `tca.log`).
- Running the `tca` analysis tool to see the coverage analysis results.

The sections below each describe options appropriate to one of these stages.

Compiling

`insure++.coverage_boolean [`**`on`**`|off]`

If set to `on`, the only data that will be stored is whether or not each block was executed. If `off`, the number of times each block was executed is also recorded. Setting this option to `on` will cause your program to compile and run slightly faster.

`insure++.coverage_map_data [`**`on`**`|off]`

If set to `on`, coverage analysis data is collected whenever applications are compiled. Such applications are then candidates for collecting coverage analysis data at runtime. Setting this option to `off` disables this. Applications must be compiled with this option on before the runtime coverage analysis options have any effect.

`insure++.coverage_map_file [filename]`

Specifies the name of the file to which the coverage analysis database will be written. Filenames may be specified using any of the standard methods that make sense at compile time. See “Filenames” on page 165. For example, you cannot use `%p` or `%D` with this option. If this option is not specified, the default filename `tca.map` is used.

Running

insure++.coverage_banner [**on**|off]

If set to on, a message is displayed at runtime indicating the file to which coverage analysis data will be written. Setting this to off disables this message.

insure++.coverage_log_data [**on**|off]

If set to on, coverage analysis data is collected whenever applications which have been compiled for coverage analysis are executed. Setting this option to off disables this.

insure++.coverage_log_file [filename]

Specifies the name of the file to which coverage analysis data will be written. Filenames may be specified using any of the standard methods. Refer to “Filenames” on page 165 for appropriate specification methods. If this option is not specified, the default filename tca.log is used.

insure++.coverage_overwrite [on|**off**]

Indicates how data from successive application runs will be merged with any existing data. If on the existing log file will be overwritten each time the application runs. If this is turned off, then each run causes new data to be added to the existing log file to form a new, combined result. In this mode, the log file data will still be discarded if the executable has changed since the last recorded log data.

insure++.coverage_switches switches

Specifies the command line arguments to be passed to the tca command when it is executed as a result of a “summarize coverage” option.

Running TCA

```
registertool TCA version
```

Used for internal maintenance. This option should not be modified.

```
tca.password arg1 arg2 arg3
```

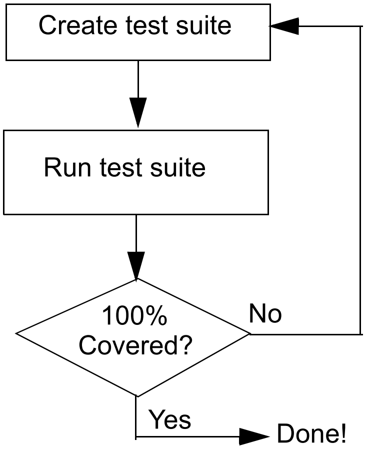
Used for internal maintenance. This option should not be added or modified by hand. Licenses should be managed with `pslic`.

Building a Test Suite

Now that you have all this coverage analysis information, what's the best way to use it? Typically, you will have several tests for your code designed to exercise various features. Together, these tests make a test suite. After you have run your tests, use TCA to discover which blocks have not been executed. This will indicate deficiencies in your test suite.

At this point, you should create more tests to try and exercise the code that was missed by your test suite so far. After you have created more tests, you can repeat the process. If the goal of 100% coverage is unreachable, you will need to make a subjective decision about how thorough you can afford to be.

This process is illustrated in the following flow diagram:


Create test suite
Run test suite
100% Covered?
No
Yes
Done!

Configuration Options

Insure++ programs read options from files called `.psrc`, which may exist at various locations in the file system. These options control the behavior of Insure++ and programs compiled with Insure++. The files are processed in the order specified below.

- The file `.psrc` in the appropriate lib and compiler subdirectories of the main Insure++ installation directory. For example:

  ```
  /usr/local/insure/lib.solaris/cc/.psrc
  ```

  or

  ```
  /usr/local/insure/lib.aix5/xlC/.psrc
  ```

- The file `.psrc` in the main installation directory.
- A file `.psrc` in your `$HOME` directory, if it exists.
- A file `.psrc` in the current working directory, if it exists.
- Files specified with the `-Zop` switch and individual options specified with the `-Zoi` switch to the insure command in the order present on the command line.

In each case, options found in later files override those seen earlier. All files mentioned above will be processed and the options set before any source files are processed.

Typically, compiler-dependent options are stored in the first location, site-dependent options are stored in the second location, user-dependent options are stored in the third location, and project-dependent options are stored in the fourth location. `-Zop` is commonly used for file-dependent options, and `-Zoi` is commonly used for temporary options.

Working on Multiple Platforms Or With Multiple Compilers

Many projects involve porting applications to several different platforms or the use of more than one compiler. Insure++ deals with this by using two built-in variables, which denote the machine architecture on which you are running and the name of the compiler you are using. Anywhere that you would normally specify a pathname or filename, you can then use these values to switch between various options, each specific to a particular machine or compiler.

In the compiler-default .psrc files, there are several interface_library options of the form

```
Insure++.InterfaceLibrary
$PARASOFT/lib.%a/%c/builtin.tqi \
$PARASOFT/lib.%a/libtqsiic%c.a
```

Despite appearances, the `PARASOFT` variable used above is not a true environment variable. If the `PARASOFT` environment variable is not set by the user, it will be expanded automatically by Insure++.

Cross-Compiler Issues

You can use the new `-Ztarget` option to support cross compiling using the gcc or g++ compilers for the `linux2`, `linux_ppc`, and `linux_mips` architectures. To use the `-Ztarget` option, follow these steps:

1. Install Insure++ 6.1 for `linux2`, `linux_ppc`, and/or `linux_mips` into the same directory.
2. Set `Insure++.Compiler <full path to your cross compiler>` in the `.psrc` file.
3. Set `Insure++.CompilerAcronym <g++/gcc>` in the `.psrc` file.

For example, to cross compile the file `hello.c` for the Linux MIPS architecture from Linux x86:

1. cd to the `../examples/c` directory:
2. At the command prompt, type:

```
insure++ -g -o hello hello.c -Ztarget linux_mips -Zoi "com-
piler <cross>" -Zoi "compilerAcronym gcc"
```

This will cross-compile the file for the Linux MIPS architecture using the x86 platform and the gcc compiler.

Option Values

The following sections describe the Insure++ configuration options. Options are divided into two classes: compile time and runtime. Modifying one of the compile time options requires that files be recompiled before it can take effect. The runtime options merely require that the program be executed again.

Some options have default values, which are printed in the following section in **boldface**.

Filenames

A number of the Insure++ options can specify filenames for various configuration and/or output files. You may either enter a simple filename or give a template which takes the form of a string of characters with tokens such as `%d`, `%p`, or `%V` embedded in it. Each of these is expanded to indicate a certain property of your program as indicated in the following tables. The first table lists the options that can be used at both compile and runtime:

Key	Meaning
`%a`	Machine architecture on which you are running. For example, `solaris`, `aix4`, `hp10`, and so on.
`%c`	Abbreviated name of the compiler you are using. For example, `cc`, `gcc`, `xlC`.
`%r`	Insure++ version number. For example, `6.1`
`%R`	Insure++ version number without periods (`.`). For example, version 6.1 becomes `61`.

This second table lists the tokens available only at runtime:

Key	Meaning
`%d`	Time of program compilation in format: `YYYYMMDDHHMMSS`
`%D`	Time of program execution in format: `YYYYMMDDHHMMSS`
`%n`	Integer sufficient to make filename unique, starting at 0
`%p`	Process I.D.
`%v`	Name of executable
`%V`	Directory containing executable

Example One

The name template report_file %v-errs.%D when executed with a program called foo at 10:30 a.m. on the 21st of December 2001, might generate a report file with the name foo-errs.20011221103032.

The last two digits are the seconds after 10:30 on which execution began. You can also include environment variables in these filenames.

Example Two

For TCA, the option coverage_map_file tca.map.%a.%c might generate a report file with the name tca.map.sun4.cc

You can also include environment variables in these filenames so that

```
        $HOME/tca/tca.map.a%.c%
```

generates the same filename as the previous example, but also ensures that the output is placed in the tca sub-directory of the user's home directory.

Advanced Configuration Options Used by Insure++

Compiling/Linking

`insure++.c_as_cpp [on|`**`off`**`]`

Specifies whether files with the `.c` extension should be treated as C++ source code. With this option `off`, Insure++ will treat files with the `.c` extension as C code only. If you use C++ code in `.c` files, you should turn this option `on`.

`insure++.checking_uninit [`**`on`**`|off]`

Specifies that the code to perform flow-analysis and check for uninitialized variables should not be inserted. Runtime uninitialized variable checking is then limited to uninitialized pointer variables. See “insure++.checking_uninit [on|off]” on page 181 for the runtime effects of this option.

`insure++.compiler compiler_name`

Specifies the name of an alternative compiler, such as `gcc`. This option overrides all other `compiler_*` options: `compiler_c`, `compiler_cpp`, and `compiler_default`. The indicated compiler will be called every time `insure++` is called.

`insure++.compiler_c C_compiler_name`

Specifies the name of the default C compiler. This compiler will be called for any `.c` files. The default is `cl`. This option is overridden by the `compiler` and `compiler_acronym` options.

insure++.compiler_cpp C++_compiler_name

Specifies the name of the default C++ compiler, such as cl. This compiler will be called for any .cc, .cpp, and .cxx. The default is platform-dependent. This option is overridden by the compiler and compiler_acronym options.

insure++.compiler_default [**c**|cpp]

Specifies whether the default C or C++ compiler should be called to link when there are no source files on the link line. This option is overridden by the compiler and compiler_acronym options.

insure++.compiler_deficient
[all|address|cast|enum|member_pointer|
scope_resolution|static_temps|struct_offset|types|
no_address|no_cast|no_enum|no_member_pointer|
no_scope_resolution|no_static_temps|no_struct_offset|
no_types|none]

This options specifies which features are not supported by your compiler. The default is compiler-dependent.

Keyword	Meaning
all	Includes all positive keywords
address/no_address	
cast/no_cast	
enum/no_enum	
member_pointer/no_member_pointer	
scope_resolution/no_scope_resolution	
static_temps/no_static_temps	
struct_offset/no_struct_offset	

Keyword	Meaning
types/no_types	
none	Compiler handles all cases

Different compilers require different levels of this option as indicated in the compiler-specific README files and in ($PARASOFT)//$compiler.

insure++.compiler_fault_recovery [off|**on**]

This option controls how Insure++ recovers from errors during compilation and linking. With fault recovery on, if there is an error during compilation, Insure++ will simply compile with the compiler only and will not process that file. If there is an error during linking, Insure++ will attempt to take corrective action by using the -Zsl option. If this option is turned off, Insure++ will make only one attempt at each compile and link.

insure++.compiler_fault_recovery_banner [**off**|on]

When activated, this option prints out a message that fault recovery has begun when instrumenting a file or when linking.

insure++.compiler_keyword [*|const|inline|signed|volatile] keyword

Specifies a new compiler keyword (by using the *) or a different name for a standard keyword. For example, if your compiler uses __const as a keyword, use the option insure++.compiler_keyword const __const.

```
insure++.compiler_options keyword value
```

Specifies various capabilities of the compiler in use, as described in the following table.

Keyword	**Value**	**Meaning**
`ansi`	None	Assumes compiler supports ANSI C (default)
`bfunc <type>`	Function name	Specifies that the given function is a “built-in” that is treated specially by the compiler. The optional `type` keyword specifies that the built-in has a return type other than `int`. Currently, only `long`, `double`, `char *`, and `void *` types are supported.
`bfuncnoeval`	Function name	Similar to `bfunc <type>`. Specifies that the given function is a "built-in" function that is treated specially by the compiler. Unlike `bfunc`, this option additionally specifies that the built-in function's arguments are not evaluated by the compiler.
`btype`	Type name	Specifies that the given type is a “built-in” that is treated specially by the compiler

Keyword	Value	Meaning
`bvar <type>`	Variable name	Specifies that the given variable is a “built-in” that is treated specially by the compiler. The optional `type` keyword specifies that the built-in has a return type other than `int`. Currently, only `long`, `double`, `char *`, and `void *` types are supported.
`esc_x`	Integer	Specifies how the compiler treats the `\x` escape sequence. Possible values are: 0 Treat `\x` as the single character `x` (Kernighan and Ritchie style). -1 Treat as a hex constant. Consume as many hex digits as possible. >0 Treat as a hex constant. Consume at most the given number of hex digits.

`for_scope`	`nested`
	`notnested`
	`optional`
		Specifies how `for(int i; ...; ...)` is scoped. Possible values are:
		nested New ANSI standard, always treat as nested.
		notnested Old standard, never treat as nested.
		optional New standard by default, but old-style code is detected and treated properly (and silently)

Keyword	Value	Meaning
knr	None	Assumes compiler uses Kernighan and Ritchie (old-style) C
loose	None	Enables non-ANSI extensions (default)
namespaces	None	Specifies that namespace is a keyword (default)
nonamespaces	None	Specifies that namespace is not a keyword
nobtype	Type name	This option is the opposite of btype. It specifies that the given type is not a "built-in" type recognized by the compiler. For example, in C++, Insure++ treats bool as a built-in type by default; CompilerOptions nobtype bool specifies that bool is not a built-in type.
promote_long	None	Specifies that integral data types are promoted to long in expressions, rather than int
sizet	d, ld, u, lu	Specifies the data type returned by the sizeof operator, as follows: d=int, ld=long, u=unsigned int, lu=unsigned long.
strict	None	Disables non-ANSI extensions (compiler dependent)

Keyword	Value	Meaning
`xfunctype`	Function name	Indicates that the named function takes an argument which is a data type rather than a variable (for example, `alignof`)

`insure++.coverage_boolean [`**`on`**`|off]`

If set to `on`, the only data that will be stored is whether or not each block was executed. If `off`, the number of times each block was executed is also recorded. Setting this option to `on` will cause your program to compile and run slightly faster.

`insure++.coverage_map_data [`**`on`**`|off]`

Prompts Insure++ to generate coverage map data for TCA.

`insure++.coverage_map_file [filename]`

Specifies the full path to the directory where Insure++ writes the `tca.map` file.

`insure++.coverage_only [on|`**`off`**`]`

Compiles and generates coverage information (`tca.map`). It also compiles and links source code for TCA.

`insure++.error_format string`

Specifies the format for error message banners generated by Insure++. The string argument will be displayed as entered with the macro substitutions taking place as shown in the following table. The string may also contain standard C formatting characters, such as `\n`. For examples, see “Customizing the Output Format” on page 68.

Key	Expands to
`%c`	Error category (and sub-category if required)
`%d`	Date on which the error occurs (`DD-MM-YYYY`)
`%f`	Filename containing the error
`%F`	Full pathname of the file containing the error
`%h`	Name of the host on which the application is running
`%l`	Line number containing the error
`%p`	Process ID of the process incurring the error
`%t`	Time at which the error occurred (`HH:MM:SS`)

`insure++.file_ignore string`

Specifies that any file which matches the string will not be processed by Insure++, but will be passed straight through to the compiler. The string should be a glob-style regular expression.

This option allows you to avoid processing files that you know are correct. This can significantly speed up execution and shrink your code.

`insure++.function_ignore file::function_name`

This option tells Insure++ not to instrument the given function (the file qualifier is optional). This is equivalent to turning off the checking for that routine. If the function in question is a bottle-neck, this may dramatically increase the runtime performance of the code processed with Insure++. `function_name` can accept the `*` wildcard.

For example, the option

```
        insure++.function_ignore foo*
```

turns off instrumentation for the functions `foo`, `foobar`, and so on.

`insure++.header_ignore string`

Specifies that any function in the filename specified by the string will not be instrumented by Insure++. The string should be a glob-style regular expression and should include the full path.

This option allows you to avoid doing runtime checking in header files that you know are correct. This can significantly speed up execution and shrink your code. Please note, however, that the file must still be parsed by Insure++, so this option will not eliminate compile time warnings and errors, only runtime checking.

`insure++.init_extension [c|cc|C|cpp|cxx|c++]`

This option tells Insure++ to use the given extension and language for the Insure++ initialization code source file. The extension can be any one of the Insure++-supported extensions: `c` (for C code) or `cc`, `cpp`, `cxx`, or `c++` (for C++ code). This option only needs to be used to override the default, which is the extension used by any source files on the `insure++` command line. If there are no source files on the command line (for example, a separate link command), Insure++ will use a c extension by default.

`insure++.interface_ignore interface name`

This options disables type checking for the specified interface at compile time. `insure++.interface_ignore` uses wildcards in the same manner as `insure++.function_ignore`.

`insure++.linker linker_name`

Specifies the name of an alternative linker. This only applies if you are using the `inslink` command.

`insure++.linker_source source_code`

This option tells Insure++ to add the given code to its initialization file. This can help eliminate unresolved symbols caused by linker bugs.

`insure++.linker_stub symbol_name`

This option tells Insure++ to create and link in a dummy function for the given `symbol_name`. This can help eliminate unresolved symbols caused by linker bugs.

`insure++.lrtCacheDir`

Specifies the directory where Insure++ will store instrumented versions of system libraries. By default, this is set to the "LRT-cache" subdirectory of the Insure++ installation directory. You may wish to set this option if Insure++ is installed on a read-only file-system, or on a file-system with limited disk space. If you change this option, you should move or copy the existing LRT cache directory, together with its contents, to its new location.

```
insure++.optionfile [filename]
```

This option specifies a file that Insure++ can use as a secondary `.psrc` file. This is helpful if you use environment variables, as you can specify one `.psrc` file for one type of system architecture, Linux for example, and another to use on a second architecture, such as Solaris. This option is also useful for specifying one `.psrc` file for a given project, and another `.psrc` file for a different project, as in the following example:

```
optionfile $HOME/.psrc.$ARCH
optionfile $PROJECT/myoptions
```

In this latter case, the standard bottom up preference is used to determine which file takes precedence over the other; that is, the `.psrc` file on bottom would be used first, then the one above.

```
insure++.pragma_ignore string
```

Any pragma which matches the string will be deleted by Insure++. The string should be a glob-style regular expression.

`insure++.rename_files [on|`**`off`**`]`

Normally, Insure++ creates an intermediate file which is passed to the compiler. In some cases, this may confuse debuggers. If this is the case, you can set this option and Insure++ will then rename the files during compilation so that they are the same. In this case, an original source file called `foo.c` would be renamed `foo.c.ins_orig` for the duration of the call to Insure++.

For example, after setting this option, the output to the screen would be:

```
***Renaming source file foo.c to foo.c.ins_orig***
```

or

```
***Restoring foo.c from foo.c.ins_orig***
```

`insure++.report_banner [`**`on`**`|off]`

Controls whether or not a message is displayed on your terminal, reminding you that error messages have been redirected to a file. See “The Report File” on page 67 for more information.

`insure++.report_file [filename|insra|`**`stderr`**`]`

Specifies the name of the report file. Environment variables and various pattern generation keys may appear in `filename`. For more information, see “Filenames” on page 165. Use of the special filename `insra` tells Insure++ to send its output to Insra.

`insure++.sizeof type value`

This option allows you to specify data type sizes which differ from the host machine, which is often necessary for cross compilation. `value` should be the number `sizeof(type)` would return on the target machine. Allowed `type` arguments are `char`, `double`, `float`, `int`, `long`, `long double`, `long long`, `short`, and `void *`.

`insure++.stack_internal [on|`**`off`**`]`

If you are using the `symbol_table off` runtime option you can set this option to `on` and recompile your program to get filenames and line numbers in stack traces without using the symbol table reader. See “insure++.symbol_table [on|off]” on page 189 for more information.

insure++.stackpc [on|**off**]

This option causes Insure++ to add to stack traces the actual address of the instruction where an error has occurred, as shown in the following partial error message:

```
Memory leaked leaving scope: p
Lost block : 0x0804b710 thru 0x0804b719 (10 bytes)
         p, allocated at leakscop.c, 9
              malloc() pc: 0x40041719 (interface)
               gimme() pc: 0x08049916 leakscop.c, 9
                main() pc: 0x08049a59 leakscop.c, 15
Stack trace where the error occurred:
               gimme() pc: 0x080499b5 leakscop.c, 10
                main() pc: 0x08049a59 leakscop.c, 15
```

insure++.string_table [on|**off**]

Moves the string table into a separate file.

insure++.suppress code

Suppresses compile time messages matching the indicated error code. Context-sensitive suppression does not apply at compile time. See “Suppressing Error Messages” on page 73 for more information.

insure++.suppress_output string

Suppresses compile time messages including the indicated error string. See “Suppressing Other Warning Messages” on page 76 for more information. For example, to suppress the warning:

```
[foo.c:5] Warning: bad conversion in assignment: char * =
int *
 >> ptr = iptr;
```

add the following string to this value:

```
bad conversion in assignment
```

insure++.temp_directory path

Specifies the directory where Insure++ will write its temporary files, for example, C:\tmp. The default is the Windows temporary directory. Setting path to a directory local to your machine can dramatically improve compile time performance if you are compiling on a remotely mounted file system.

insure++.uninit_flow [1|2|3|...|**100**|...|1000]

Insure++ can perform a lot of checks for uninitialized memory at compile time. This value specifies how hard Insure++ should try to analyze this at compile time. A high number will make Insure++ run slower at compile time, but will produce a faster executable. Values over 1000 are not significant except for very complicated functions.

insure++.unsuppress code

Enables compile time messages matching the indicated error code. Context sensitive suppression is not supported at compile time. See “Enabling Error Messages” on page 77 for more information.

insure++.virtual_checking [**on**|off]

Specifies whether VIRTUAL_BAD error messages will be generated. See “VIRTUAL_BAD” on page 202 for more information about this error message.

Running

`insure++.checking_uninit [`**`on`**`|off]`

If set to `off`, this option specifies that the code to perform flow-analysis and checking for uninitialized variables should not be executed, if present. See "Compiling/Linking" on page 167 for the compile time effects of this option. Runtime uninitialized variable checking is then limited to uninitialized pointer variables.

`insure++.checking_uninit_min_size [1|`**`2`**`|3|...]`

Specifies the minimum size in bytes of data types on which Insure++ should perform full uninitialized memory checking. The default is `2`, which means that `char`s will not be checked by default. Setting this option to `1` will check `char`s, but may result in false errors being reported. These can be eliminated by using the `checking_uninit_pattern` option to change the pattern used (see below).

`insure++.checking_uninit_pattern pattern`

Specifies the pattern to be used by the uninitialized memory checking algorithm. The default is `deadbeef`. `pattern` must be a valid, 8-digit hexadecimal value.

`insure++.coverage_banner [`**`on`**`|off]`

Prompts Insure++ to display a message about coverage information written to `tca.log`.

`insure++.coverage_log_file [filename]`

Specifies the full path to the directory where Insure++ writes the `tca.log` file.

`insure++.coverage_overwrite [on|`**`off`**`]`

Determines if the `tca.log` file will be overwritten on each run.

`insure++.demangle [off|`**`on`**`|types|full_types]`

Specifies the level of function name demangling in reports generated by Insure++. If you have a function

```
void func(const int)
```

you will get the following results:

Keyword	**Result**
`off`	`func__FCi`
`on`	`func`
`types`	`func(int)`
`full_types`	`func(const int)`

`insure++.demangle_method [external <filtname>|CC|gcc]`

Specifies compiler-specific algorithm for demangling function names. Currently supported compiler algorithms are C-Front based C++ compilers (for example, CC). If you are using a different compiler, Insure++ understands most other demangling formats as well. The filter <filtname> option allows the use of the external demangler filtname. The default is compiler-dependent. See the compiler level .psrc file, which is in the directory lib.$ARCH/$COMPILER.

This option is a compiled-in option, so you will need to prepend a ! to the option in the `.psrc` file to change the setting at runtime.

`insure++.error_format string`

Specifies the format for error message banners generated by Insure++. The string argument is displayed as it is entered, with the macro substitutions taking place as shown in the `compiler_deficient` table. The string may also contain standard C formatting characters, such as `\n`. For examples, see “Customizing the Output Format” on page 68.

`insure++.exit_hook [`**`on`**`|off]`

Normally, Insure++ uses the appropriate `atexit`, `onexit`, or `on_exit` function call to perform special handling at exit. If for some reason, this is a problem on your system, you can disable this functionality via the `exit_hook` option.

`insure++.exit_on_error [`**`0`**`|1|2|3|...]`

Causes the user program to quit (with non-zero exit status) after reporting the given number of errors. The default is `0`, which means that all errors will be reported and the program will terminate normally.

`insure++.exit_on_error_banner [`**`on`**`|off]`

Normally, when Insure++ causes your program to quit due to the `exit_on_error` option, it will print a brief message like the following:

```
** User selected maximum error count reached: 10. Program
exiting.**
```

Setting this option to `off` will disable this message.

`insure++.free_delay [0|1|2|3|...|`**`119`**`|...]`

This option controls how long the Insure++ runtime holds onto `free`’d blocks before allowing them to be reused. This is not necessary for error detection, but can be useful in modifying the behavior of your program for stress-testing. The number represents how many freed blocks are held back at a time--large numbers limit memory reuse, and `0` maximizes memory reuse.

`insure++.free_pattern pattern`

Specifies a pattern that will be written on top of memory whenever it is freed. This pattern will be repeated for each byte in the freed region. The default is `0`, which means no pattern will be written.

Note: On some systems whose libraries assume freed memory is still valid, this may cause your program to crash.

`insure++.GusCacheDir`

Specifies the directory where Insure++ will store its cache files containing symbolic debugging information. By default, this is set to the "GUS-cache" subdirectory of the Insure++ installation directory. You may wish to set this option if Insure++ is installed on a read-only file-system, or a on file-system with limited disk space. You may safely delete the contents of the GUS cache directory when you are not actively using Insure++ (its contents will be automatically regenerated the next time you use Insure++).

`insure++.ignore_wild [on|`**`off`**`]`

Specifies whether Insure++ will do checking for wild pointers. Turning this option `on` turns off wild pointer checking.

`insure++.leak_combine [none|`**`trace`**`|location]`

Specifies how to combine leaks for the memory leak summary report. Combining by `trace` means all blocks allocated with identical stack traces will be combined into a single entry. Combining by `location` means all allocations from the same file and line (independent of the rest of the stack trace) will be combined. none means each allocation will be listed separately.

`insure++.leak_search [`**`on`**`|off]`

Specifies additional leak checking at runtime before a leak is reported. Requires that the symbol table reader be turned on.

`insure++.leak_sort [none|frequency|location|`**`size`**`]`

Specifies by what criterion the memory leak summary report is sorted. Setting this to `none` may provide better performance at exit if you have many leaks.

`insure++.leak_summary_filter`

Controls which blocks are reported in the “leaks detected at exit” and “outstanding” sections of the leak summary. For example,

```
        insure++.leak_summary_filter *main
```

restricts the leak summary to those blocks with stack traces ending in `main`.

```
        insure++.leak_summary_filter -! * Tcl_Alloc* *
```

filters out the following leak summary entries:

```
3607 bytes 3 chunks allocated
malloc()
Tcl_Alloc()
Tcl_NewStringObj()
Tcl_EvalTokens()
Tcl_EvalEx()
Tcl_EvalFile()
TclExecuteByteCode()
Tcl_EvalObjEx()
Tcl_UplevelObjCmd()
```

`insure++.leak_sweep [`**`on`**`|off]`

Specifies additional leak checking at the termination of the program. Requires that the symbol table reader be turned on.

insure++.leak_trace [**on**|off]

This option determines whether or not full stack traces will be shown in the memory leak summary report.

insure++.new_overhead [0|2|4|6|8|...]

Specifies the number of bytes allocated as overhead each time new[] is called. The default is compiler-dependent, but is typically 0, 4, or 8.

insure++.optionfile [filename]

This option is described in the Compiling/Linking section of the Configuration Options. It can be used either during runtime or when compiling and linking a file or files.

insure++.pointer_slack [0|**1**|2]

This controls a heuristic in Insure++. When a pointer does not point to a valid block, but does point to an area 1 byte past the end of a valid block, does the pointer really point to that block? The value of this argument controls Insure++'s answer. The default should be changed only if Insure++ is not working correctly on your program.

Value	Meaning
0	Never assume the pointer points to the previous block
1	Assume the pointer points to the previous block if that block was dynamically allocated
2	Always assume the pointer points to the previous block. This tends to be incorrect for stack and global variables, since they are usually adjacent in memory

`insure++.report_banner [`**`on`**`|off]`

Controls whether or not a message is displayed on your terminal, reminding you that error messages have been redirected to a file. For more information, see “Filenames” on page 165.

`insure++.report_file [filename|Insra|`**`stderr`**`]`

Specifies the name of the report file. Environment variables and various pattern generation keys may appear in `filename`. For more information, see “Filenames” on page 165. Use of the special filename `Insra` tells Insure++ to send its output to Insra.

`insure++.report_limit [-1|0|`**`1`**`|2|3|...]`

Displays only the first given number of errors of each type at any particular source line. Setting this option to `-1` will show all errors. Setting it to `0` will only show errors in summary reports, and not at runtime. See “Displaying Repeated Errors” on page 70 for more information.

`insure++.report_overwrite [`**`on`**`|off]`

If set to `off`, error messages are appended to the report file rather than overwriting it on each run.

`insure++.runtime [`**`on`**`|off]`

If set to `off`, no runtime checking or profiling is performed. The program will run much faster this way. This option can be used to check if a particular fix has cured a problem, without recompiling the application without Insure++.

```
insure++.source_path .:/users/boswell/src:/src
```

This option takes a list of directories in which to search for source files See “Searching For Source Code” on page 72 for more information. This will only be necessary if your source code has moved since it was compiled, as Insure++ remembers where all your source files are located.

```
insure++.summarize [bugs] [coverage] [leaks] [outstanding]
```

Generates a summary report of errors. For more information, see the following links:

- “Report Summaries” on page 77.
- “The Leak Summaries” on page 80.
- “The Coverage Summary” on page 83.

In the latter case, the `coverage_switches` option is consulted to decide how to present coverage data; see “Options Used By TCA” on page 190 for more information. The `leaks` and `outstanding` reports are affected by the `leak_combine`, `leak_sort`, and `leak_trace` options. With no arguments, this option will summarize the `bugs` and `leaks` summaries.

Note: This option has changed slightly in versions 3.1 and higher. The old leak defaults are equivalent to `leak_combine location`, `leak_sort location`, `leak_trace off`. The old `detailed` option is replaced by `leak_trace on`.

```
insure++.summarize_on_error [0|1|2|3|...]
```

Specifies how many errors must be generated before a summary (if requested) is printed. The default is `0`, which means that summaries are always printed on demand. If the number is `1` or higher, summaries are only printed if *at least* the given number of bugs (or leaks) occurs. Suppressed errors do not count towards this number. If no argument is given with this option, a value of `1` is assumed.

`insure++.suppress code [{context}]`

Suppress error messages matching the given error code and occurring in the (optionally) specified context. See “Suppressing Error Messages” on page 73 for more information.

`insure++.symbol_banner [on|`**`off`**`]`

If set, Insure++ displays a message indicating that the program’s symbol table is being processed whenever an application starts.

`insure++.symbol_table [`**`on`**`|off]`

If set to `on`, Insure++ will read the executable symbol table at startup. This enables Insure++ to generate full stack traces for third party libraries as well as for code compiled with Insure++. If this option is turned off, the stack traces will show only functions compiled with Insure++, but the application will use less dynamic memory and be faster on startup. To get filenames and line numbers in stack traces with this option off, you must compile your program with the `stack_internal on` option. For more information, see “Compiling/Linking” on page 167.

`insure++.trace [on|`**`off`**`]`

Turns program tracing on and off. In order to get file names and line numbers in the trace output, you must have the `stack_internal on` option set when compiling the program. See “Tracing” on page 108 for more information about program tracing.

`insure++.trace_banner [`**`on`**`|off]`

Specifies whether to print message at runtime showing file to which the trace output will be written.

`insure++.trace_file [filename|`**`stderr`**`]`

Specifies the name of the file to which the trace output will be written. `filename` may use the same special tokens shown on “Filenames” on page 165.

`insure++.unsuppress code [{context}]`

Enables error messages matching the given error codes and occurring in the (optionally) specified context. See “Suppressing Error Messages” on page 73 for more information.

Options Used By Insra

For options used by Insra, see “Insra” on page 84.

Options Used By TCA

For options used by TCA, see “Working With TCA” on page 138.

Memory Overflow

One of the common errors that Insure++ detects occurs when a program reads or writes beyond the bounds of a valid memory area. This type of problem normally generates a READ_OVERFLOW or WRITE_OVERFLOW error which describes the memory regions being accessed with their addresses and sizes as shown below.

```
[hello.c:15] **WRITE_OVERFLOW**
>>         strcat(str, argv[i]);

  Writing overflows memory: <argument 1>

          bbbbbbbbbbbbbb
          |     16     | 4 |
          wwwwwwwwwwwwwwwwww

   Writing  (w) : 0xbfffefb0 thru 0xbfffefc3 (20 bytes)
   To block (b) : 0xbfffefb0 thru 0xbfffefbf (16 bytes)
                 str, declared at hello.c, 11

  Stack trace where the error occurred:
                          strcat()  (interface)
                            main()  hello.c, 15

**Memory corrupted.  Program may crash!!**
```

Overflow Diagrams

The textual information above describes the memory blocks involved in the overflow operation using their memory addresses and sizes.

To gain a more intuitive understanding of the nature of the problem, a text-based “overflow diagram” is also shown. This pattern attempts to demonstrate the nature and extent of the problem by representing the memory blocks involved pictorially.

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbb
|          16              |   2   |
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

In this case, the row of `b` characters represents the available memory block, while the row of `w`'s shows the range of memory addresses being written. The block being written is longer than the actual memory block, which causes the error.

The numbers shown indicate the size, in bytes, of the various regions and match those of the textual error message.

The relative length and alignment of the rows of characters is intended to indicate the size and relative positioning of the memory blocks which cause the error. The above case shows both blocks beginning at the same position with the written block extending beyond the end of the memory region. If the region being written extended both before and after the available block, a diagram such as the following would have been displayed.

```
       bbbbbbbbbbbbbbbbbbbbbbbbbbbb
|  5   |          16               |   2  |
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

Completely disjointed memory blocks are indicated by a diagram of the form

```
                             bbbbbbbbbbbbbbbbbb
|  4  |         40           |       16        |
wwwwwww
```

Similar diagrams appear for both `READ_OVERFLOW` and `WRITE_OVERFLOW` errors. In the former case, the block being read is represented by a row of `r` characters instead of `w`'s. Similarly, the memory regions involved in parameter size mismatch errors are indicated using a row of `p` characters for the parameter block. See “PARM_BAD_RANGE” on page 283 for more information.

Insure++ API

This section lists the Insure++ functions that can be called from an application program.

If you are inserting these routines into your source code, you might want to use the pre-processor symbol `__INSURE__` so that they will only be called when compiling with the appropriate tools. For example:

```
void grind_away() {
#ifdef __INSURE__
         _Insure_checking_enable(0);
                  //disables Insure++ checking
#endif
              ... code ...

#ifdef __INSURE__
         _Insure_checking_enable(1);
                  //enables Insure++ checking
#endif
}
```

In this way you can use the same source code when compiling with or without Insure++.

You will also need to add prototypes for these functions to your code, particularly if you are calling these C functions from C++ code. Make sure that in this case your prototype is properly marked `extern "C"`.

Control Routine

This routine affects the behavior of Insure++ and is normally called from within your source code.

```
void _Insure_printf(char *fmt,[,arg...]);
```

Causes Insure++ to add the given character string to its output.

Memory Block Description Routines

- `size_t _Insure_list_allocated_memory(unsigned int mode)`

  Prints the total number of allocated blocks and the total number of bytes allocated. If `mode` is set to `2`, Insure++ lists all allocated memory blocks and their sizes. If `mode` is set to `1`, Insure++ lists only newly allocated or re-allocated blocks (blocks allocated or re-allocated since the last time this function was called). If `mode is` set to `0`, Insure++ does not list any blocks; it only prints the total allocation. This function returns the total number of bytes allocated.

- `void _Insure_mem_info(void *ptr);`

  Displays all information known about the memory block whose address is `ptr`. For example, the following code

```
#include <stdlib.h>

main()
{
    char *p, buf[128];

    p = malloc(100);
#ifdef __INSURE__
    _Insure_mem_info(buf);
    _Insure_mem_info(p);
#endif
    ...
```

might generate the following output

```
Pointer  : 0xf7fff74c (stack)
Offset   : 0 bytes
In block : 0xf7fff74c thru 0xf7fff7cb (128 bytes)
           buf, declared at foo.c, 4
Pointer  : 0x00024b98 (heap)
Offset   : 0 bytes
In block : 0x00024b98 thru 0x00024bfb (100 bytes)
           p, allocated at foo.c, 6
```

- `void _Insure_ptr_info(void **ptr);`

  Displays all information about the pointer whose address is passed. For example, the code

```
#include <stdlib.h>

main()
{
    char *p, buf[128];

    p = malloc(100);
#ifdef __INSURE__
    _Insure_ptr_info(&p);
#endif
     ...
```

might generate the following output

```
Pointer  : 0x00024b98 (heap)
Offset   : 0 bytes
In block : 0x00024b98 thru 0x00024bfb (100 bytes)
           p, allocated at foo.c, 6
```

- `void _Insure_leak_summary()`

  Prompts Insure++ to report a leak summary at various points during execution. If the `leak_sweep` option is enabled, this function prompts Insure++ to mark and sweep all heap blocks.

  Note: `leak_sweep` should always be enabled when using this function). If the `summarize leaks` and/or `summarize outstanding` options are set, this function prompts Insure++ to report a leak summary. It also notifies Inuse of all leaked heap blocks.

  This function may be called repeatedly throughout a process's lifetime. This is useful for monitoring leaks in a continuously running process (for example, a server) when the `leak_search` option has been disabled to improve performance.

Tracing

`_Insure_trace_annotate()` and `_Insure_trace_enable()` can be used to perform tracing. For details on tracing, see "Tracing" on page 108.

Error Codes

The following sections are intended to provide a reference for the various error messages generated by Insure++.

The table below lists each error code alphabetically together with its interpretation and an indication of whether or not it is suppressed by default. **Note:** Errors can be unsuppressed using the Suppressions Control Panel.

The following sections provide a detailed description of each error including:

- A brief explanation of what problem has been detected.
- An example program that generates a similar error.
- Output that would be generated by running the example, with annotations indicating what the various pieces of the diagnostic mean and how they should be interpreted in identifying your own problems.

  Note that the exact appearance of the error message might depend heavily on how Insure++ is currently configured.
- A brief description of ways in which the problem might be eliminated.

Note: Sometimes you will see values identified as `<argument #>` or `<return>` instead of names from your program. In this case, `<argument n>` refers to the `n`th argument passed to the current function (i.e. the one where the error was detected), and `<return>` refers to a value returned from the function indicated.

Code	Meaning	Enabled?
`ALLOC_CONFLICT`	Mixing malloc/free with new/delete	Y
(badfree)	Free called on block allocated with new	Y

Code	Meaning	Enabled?
(baddelete)	Delete `called` on block allocated with malloc	Y
`BAD_CAST`	Cast of pointer loses precision	Y
`BAD_DECL`	Incompatible global declarations	Y
`BAD_FORMAT`	Mismatch in format specification	N
(sign)	`int` vs. `unsigned int`	N
(compatible)	`int` vs. `long`, both same size	N
(incompatible)	`int` vs. `double`	Y
(other)	Wrong number of arguments	Y
`BAD_INTERFACE`	Declaration of function in interface conflicts with declaration in program	Y
`BAD_PARM`	Mismatch in argument type	N
(sign)	`int` vs. `unsigned int`	N
(compatible)	`int` vs. `long`, both same size	N
(incompatible)	`int` vs. `double`	Y
(pointer)	All pointers are equivalent	Y
(union)	Require exact match on `union`s	Y
(other)	Wrong number of arguments	Y
`COPY_BAD_RANGE`	Attempt to copy out-of-range pointer	N

Code	Meaning	Enabled?
COPY_DANGLING	Attempt to copy dangling pointer	N
COPY_UNINIT_PTR	Attempt to copy uninitialized pointer	N
COPY_WILD	Attempt to copy wild pointer	N
DEAD_CODE	Code is not evaluated, has no effect, or is unreachable	N
(emptyloopbody)	Loop body is empty	N
(emptystmt)	Statement is empty	N
(noeffect)	Code has no effect	N
(notevaluated)	Code is not evaluated	N
DELETE_MISMATCH	Mismatch between new/new[] and delete/delete[]	N
(bracket)	new, delete[]	Y
(nobracket)	new[], delete	Y
EXPR_BAD_RANGE	Expression exceeded range	N
EXPR_DANGLING	Expression uses dangling pointer	N
EXPR_NULL	Expression uses NULL pointer	Y
EXPR_UNINIT_PTR	Expression uses uninitialized pointer	Y
EXPR_UNRELATED_PTRCMP	Expression compares unrelated pointers	Y
EXPR_UNRELATED_PTRDIFF	Expression subtracts unrelated pointers	Y

Code	Meaning	Enabled?
EXPR_WILD	Expression uses wild pointer	N
FREE_BODY	Freeing memory block from body	Y
FREE_DANGLING	Freeing dangling pointer	Y
FREE_GLOBAL	Freeing global memory	Y
FREE_LOCAL	Freeing local memory	Y
FREE_UNINIT_PTR	Freeing uninitialized pointer	Y
FREE_WILD	Freeing wild pointer	Y
FUNC_BAD	Function pointer is not a function	Y
FUNC_NULL	Function pointer is NULL	Y
FUNC_UNINIT_PTR	Function pointer is uninitialized	Y
INSURE_ERROR	Internal error	Y
INSURE_WARNING	Output from iic_warning	N
LEAK_ASSIGN	Memory leaked due to pointer reassignment	Y
LEAK_FREE	Memory leaked freeing block	Y
LEAK_RETURN	Memory leaked by ignoring return value	Y
LEAK_SCOPE	Memory leaked leaving scope	Y
PARM_BAD_RANGE	Array parameter exceeded range	Y
PARM_DANGLING	Array parameter is dangling pointer	Y

Code	Meaning	Enabled?
`PARM_NULL`	Array parameter is `NULL`	Y
`PARM_UNINIT_PTR`	Array parameter is uninitialized pointer	Y
`PARM_WILD`	Array parameter is wild	Y
`READ_BAD_INDEX`	Reading array out of range	Y
`READ_DANGLING`	Reading from a dangling pointer	Y
`READ_NULL`	Reading `NULL` pointer	Y
`READ_OVERFLOW`		N
(normal)	Reading overflows memory	Y
(nonull)	String is not `NULL`-terminated within range	Y
(string)	Alleged string does not begin within legal range	Y
(struct)	Structure reference out of range	Y
(maybe)	Dereferencing structure of improper size (may be o.k.)	N
`READ_UNINIT_MEM`	Reading uninitialized memory	N
(copy)	Copy from uninitialized region	N
(read)	Use of uninitialized value	Y
`READ_UNINIT_PTR`	Reading from uninitialized pointer	Y
`READ_WILD`	Reading wild pointer	Y

Code	Meaning	Enabled?
`RETURN_DANGLING`	Returning pointer to local variable	Y
`RETURN_FAILURE`	Function call returned an error	N
`RETURN_INCONSISTENT`	Function returns inconsistent value	N
(level 1)	No declaration, returns nothing	N
(level 2)	Declared `int` returns nothing	Y
(level 3)	Declared non-`int`, returns nothing	Y
(level 4)	Returns different types at different statements	Y
`UNUSED_VAR`	Unused variables	N
(assigned)	Assigned but never used	N
(unused)	Never used	N
`USER_ERROR`	User generated error message	Y
`VIRTUAL_BAD`	Error in runtime initialization of virtual functions	Y
`WRITE_BAD_INDEX`	Writing array out of range	Y
`WRITE_DANGLING`	Writing to a dangling pointer	Y
`WRITE_NULL`	Writing to a `NULL` pointer	Y
`WRITE_OVERFLOW`		N
(normal)	Writing overflows memory	Y
(struct)	Structure reference out of range	Y

Code	Meaning	Enabled?
(maybe)	Dereferencing structure of improper size (may be o.k.)	N
`WRITE_UNINIT_PTR`	Writing to an uninitialized pointer	Y
`WRITE_WILD`	Writing to a wild pointer	Y

ALLOC_CONFLICT

Memory Allocation Conflict

This error is generated when a memory block is allocated with `new (malloc)` and freed with `free (delete)`.

Insure++ distinguishes between the two possibilities as follows:

- `badfree` - Memory was allocated with `new` or `new[]` and an attempt was made to free it with `free`.
- `baddelete` - memory was allocated with `malloc` and an attempt was made to free it with `delete` or `delete[]`.

Some compilers do allow this, but it is not good programming practice and could be a portability problem.

Problem #1

The following code shows a typical example of allocating a block of memory with `new` and then freeing it with `free`, instead of `delete`.

```
1:       /*
2:        * File: alloc1.cpp
3:        */
4:       #include <stdlib.h>
5:
6:       int main() {
7:               char *a;
8:
9:               a = new char;
10:              free(a);
11:              return 0;
12:      }
```

Diagnosis (at runtime)

```
1 [alloc1.cpp:10] **ALLOC_CONFLICT**
  >>             free(a);

2     Memory allocation conflict: a
```

```
3     free() used to deallocate memory which was allocated
            using new
                  a, allocated at:
                  main()alloc1.cpp, 9

4 Stack trace where the error occurred:
                  main()alloc1.cpp, 10
```

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Description of the conflicting allocation/deallocation.
4. Stack trace showing the function call sequence leading to the error.

Problem #2

The following code shows another typical example of this type of error, allocating a block of memory with `malloc` and then freeing it with `delete`.

```
1:        /*
2:         * File: alloc2.cpp
3:         */
4:        #include <stdlib.h>
5:
6:        int main() {
7:                 char *a;
8:
9:                 a = (char *) malloc(1);
10:                delete a;
11:                return 0;
12:       }
```

Diagnosis (at runtime)

```
1  [alloc2.cpp:10] **ALLOC_CONFLICT**
  >>             delete a;

2     Memory allocation conflict: a
```

```
3     delete operator used to deallocate memory not
          allocated by new
              block allocated at:
                  malloc()(interface)
                    main()alloc2.cpp, 9

4     Stack trace where the error occurred:
                    main()alloc2.cpp, 10
```

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Description of the conflicting allocation/deallocation.
4. Stack trace showing the function call sequence leading to the error.

Repair

This type of error can be corrected by making sure that all your memory allocations match up.

BAD_CAST

Cast of Pointer Loses Precision

Porting code between differing machine architectures can be difficult for many reasons. A particularly tricky problem occurs when the sizes of data objects, particularly pointers, differ from that for which the software was created. This error occurs when a pointer is cast to a type with fewer bits, causing information to be lost, and is designed to help in porting codes to architectures where pointers and integers are of different lengths.

Note that compilers will often catch this problem unless the user has “carefully” added the appropriate typecast to make the conversion “safe”.

Problem

The following code shows a pointer being copied to a variable too small to hold all its bits.

```
1:      /*
2:       * File: badcast.c
3:       */
4:      main()
5:      {
6:              char q, *p;
7:
8:              p = "Testing";
9:              q = (char)p;
10:             return 0;
11:     }
```

Diagnosis (during compilation)

```
1 [badcast.c:9] **BAD_CAST**
2     Cast of pointer loses precision: (char) p
  >>      q = (char) p;
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

Repair

This error normally indicates a significant portability problem that should be corrected by using a different type to save the pointer expression. In ANSI C the type `void *` will always be large enough to hold a pointer value.

BAD_DECL

Global Declarations Are Inconsistent

This error is generated whenever Insure++ detects that a variable has been declared as two different types in distinct source files. This can happen when there are two conflicting definitions of an object or when an `extern` reference to an object uses a different type than its definition.

In any case, Insure++ proceeds as though the variable *definition* is correct, overriding the `extern` reference.

Problem

In the following example, the file `baddecl1.c` declares the variable `a` to be a pointer,

```
/*
 * File: baddecl1.c
 */
int *a;
```

while the file `baddecl2.c` declares it to be an array type.

```
/*
 * File: baddecl2.c
 */
extern int a[];

main()
{
        a[0] = 10;
        return (0);
}
```

Diagnosis (at runtime)

```
  [baddecl2.c:4] **BAD_DECL**
1 >>                extern int a[];

2     Incompatible global declarations: a
```

```
3     Array and non-array declarations are not equivalent.
      Actual declaration:
          non-array (4 bytes),declared at baddecl1.c, 4
4     Conflicting declaration:
          array of unspecified size,
                  declared at baddecl2.c, 4
```

1. Source line at which the problem was detected.
2. Description of the problem and the object whose declarations conflict.
3. Brief description of the conflict.
4. Information about the conflicting definitions, including the sizes of the declared objects and the locations of their declarations.

Repair

The lines on which the conflicting declarations are made are both shown in the diagnostic report. They should be examined and the conflict resolved.

In the case shown here, for example, a suitable correction would be to change the declaration file to declare an array with a fixed size, e.g.,

```
baddecl1.c, 4:            int a[10];
```

An alternative correction would be to change the definition in `baddecl2.c` to indicate a pointer variable, e.g.,

```
baddecl2.c, 4:            extern int *a;
```

Note that this change on its own will not fix the problem. In fact, if you ran the program modified this way, you would get another error, `EXPR_NULL`, because the pointer `a` doesn't actually point to anything and is `NULL` by virtue of being a global variable, initialized to zero.

To make this version of the code correct, you would need to include something to allocate memory and store the pointer in `a`. For example,

```
/*
 * File: baddecl2.c (modified)
 */
#include <stdlib.h>
extern int *a;
```

```

main()
{
        a = (int *)malloc(10*sizeof(int));
        a[0] = 10;
}
```

Some applications may genuinely need to declare objects with different sizes, in which case you can suppress error messages by suppressing `BAD_DECL` in the Suppressions Control Panel.

BAD_FORMAT

Mismatch In Format Specification

This error is generated when a call to one of the printf or scanf routines contains a mismatch between a parameter type and the corresponding format specifier or the format string is nonsensical.

Insure++ distinguishes several types of mismatches which have different levels of severity as follows:

- sign - Types differ only by sign, e.g., int vs. unsigned int.
- compatible - Fundamental types are different but they happen to have the same representation on the particular hardware in use, e.g., int vs. long on machines where both are 32-bits, or int * vs. long where both are 32-bits.
- incompatible - Fundamental types are different, e.g. int vs. double.
- other - A problem other than an argument type mismatch is detected, such as passing the wrong number of arguments.

Error messages are classified according to this scheme and can be selectively enabled or disabled as described in the section “Repair” on page 216.

Problem #1

An example of format type mismatch occurs when the format specifiers passed to one of the printf routines do not correspond to the data, as shown below.

```
/*
 * File: badform1.c
 */
main()
{
        double f = 1.23;
        int i = 99;

        printf("%d %f\n”, f, i);
}
```

This type of mismatch is detected during compilation.

Diagnosis (during compilation)

```
1 [badform1.c:9] **BAD_FORMAT(incompatible)**
2        Wrong type passed to printf (argument 2).
         Expected int, found double.
>>                printf("%d %f\n", f, i);

[badform1.c:9] **BAD_FORMAT(incompatible)**
         Wrong type passed to printf (argument 3).
         Expected double, found int.
>>                printf("%d %f\n", f, i);
```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

Problem #2

A more dangerous problem occurs when the types passed as arguments to one of the `scanf` functions are incorrect. In the following code, for example, the call to `scanf` tries to read a double precision value, indicated by the `%lf` format, into a single precision value. This will overwrite memory.

```
/*
 * File: badform2.c
 */
main()
{
        int a;
        float f;

        scanf("%lf”, &f);
}
```

This problem is again diagnosed at compile time (along with the `WRITE_OVERFLOW`, which is not shown below).

Diagnosis (during compilation)

```
1 [badform2.c:9] **BAD_FORMAT(incompatible)**
2        Wrong type passed to scanf (argument 2).
         Expected double *, found float *.
>>                scanf("%lf\n", &f);
```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

Problem #3

A third type of problem is caused when the format string being used is a variable rather than an explicit string. The following code contains an error handler that attempts to print out a message containing a filename and line number. In line 18 of the calling routine, however, the arguments are reversed.

```
1:       /*
2:        * File: badform3.c
3:        */
4:       char *file;
5:       int line;
6:
7:       error(format)
8:                char *format;
9:       {
10:              printf(format, file, line);
11:      }
12:
13:      main()
14:      {
15:              file = "foo.c";
16:              line = 3;
17:
18:              error("Line %d, file %s\n");
19:      }
```

Diagnosis (at runtime)

```
[badform3.c:10] **BAD_FORMAT(incompatible)**
1 >>              printf(format, file, line);

2        Format string is inconsistent:
                  Wrong type passed to printf (argument 3).
         Expected pointer, found int.
3        Format string: "Line %d, file %s\n"
         Stack trace where the error occurred:
4                 error() badform3.c, 10
                  main() badform3.c, 18
```

1. Source line at which the problem was detected.
2. Description of the problem and the argument that is in error.
3. Explanation of the error and the format string that caused it.
4. Stack trace showing the function call sequence leading to the error.

The error diagnosed in this message is in the `incompatible` category, because any attempt to print a string by passing an integer variable will result in garbage. Note that with some compilers, this program may cause a core dump because of this error, while others will merely produce incorrect output.

There is, however, a second potential error in this code in the same line.

Because the arguments are in the wrong order in line 7, an attempt will be made to print a pointer variable as an integer. This error is in the `compatible` class, since a pointer and an integer are both the same size in memory. Since `compatible` `BAD_FORMAT` errors are suppressed by default, you will not see it. (These errors are suppressed because they tend to cause unexpected rather than incorrect behavior.)

If you enabled these errors, you would see a second problem report from this code.

Note: If you run Insure++ on an architecture where pointers and integers are not the same length, then this second error would also be in the `incompatible` class and would be displayed by default.

Repair

Most of these problems are simple to correct based on the information given. Normally, the correction is one or more of the following

- Change the format specifier used in the format string.
- Change the type of the variable involved.
- Add a suitable typecast.

For example, problem #1 can be corrected by simply changing the incorrect line of code as follows

```
badform1.c, line 9:printf("%d %f\n", i, f);
```

The other problems can be similarly corrected.

If your application generates error messages that you wish to ignore, you can suppress `BAD_FORMAT` in the Suppressions Control Panel.

This directive suppresses all `BAD_FORMAT` messages. If you wish to be more selective and suppress only a certain type of error, you can use the syntax

```
BAD_FORMAT(class1, class2, …)
```

where the arguments are one or more of the identifiers for the various categories of errors described in “Mismatch In Format Specification” on page 212.

Similarly, you can enable suppressed types by unsuppressing them in the Suppressions Control Panel. The problem with the pointer and integer that was not shown in the current example could be displayed by unsuppressing `BAD_FORMAT(compatible)` in the Suppressions Control Panel. For an example of this option, as well as the remaining subcategories of `BAD_FORMAT`, see the example `badform4.c`.

BAD_INTERFACE

Function Declarations Conflict with Interface

Actual declaration of xxx conflicts with interface, or ignoring interface for xxx conflicts with static or in-line declaration. This error will be generated any time there is a significant discrepancy between the source code being processed and an interface to one of the functions in the code. Common sources of this problem are redeclarations of standard system functions in your code.

Problem

The following code shows a redeclaration of the function `printf` which will conflict with the version of the function expected by the interface.

```
/*
 * File: badint.c
 */
#include <stdio.h>

static void printf(i)
         int i;
{
         fprintf(stdout, “%d\n”, i);
}
```

Diagnosis (during compilation)

```
1 [badint.c:6] **BAD_INTERFACE**
2        Ignoring interface for printf: conflicts with
         static or inline declaration.
>>          static void printf(i)
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

Repair

There are several ways to approach solving this problem. The correct solution for your situation depends upon why the function was redefined in your code. If this is a version of the function that is used with all of your code, a permanent solution would be to write a new interface corresponding to your version of the function. A quicker, more temporary solution, appropriate if you only use this version of the function occasionally, would be to temporarily disable the checking of this interface using the `interface_ignore` option. This option can be turned on and off on a per file basis as you work with different code which uses different versions of the function in question.

BAD_PARM

Mismatch In Argument Type

This error is generated when an argument to a function or subroutine does not match the type specified in an earlier declaration or an interface file.

Insure++ distinguishes several types of mismatch which have different levels of severity as follows:

- `sign`: Types differ only by sign, e.g., `int` vs. `unsigned int`.
- `compatible`: Fundamental types are different but they happen to have the same representation on the particular hardware in use, e.g., `int` vs. `long` on machines where both are 32-bits.
- `incompatible`: Fundamental types are different, e.g. `int` vs. `float`.
- `union`: Forces a declared union argument to match only a similar union as an actual argument. If this is suppressed, you may pass any of the individual union elements to the routine, rather than the union type, or pass a union to a routine which expects one of the union-elements as an argument.
- `other`: An error was detected that is not simply a mismatched argument type, such as passing the wrong number of arguments to a function.
- `pointer`: This is not an error class, but a keyword used to suppress messages about mismatched pointer types, such as `int *` vs. `char *`. See “Repair” on page 223.

Error messages are classified according to this scheme and can be selectively enabled or disabled as described in the section “Repair” on page 223.

Problem #1

The following shows an error in which an incorrect argument is passed to the function `foo`.

```
1:       /*
2:        * File: badparm1.c
3:        */
4:       void foo(str)
5:                char *str;
6:       {
7:                return;
8:       }
9:
10:      main()
11:      {
12:               int *iptr;
13:
14:               foo(iptr);
15:               return (0);
16:      }
```

This type of mismatch is detected during compilation.

Diagnosis (during compilation)

```
1 [badparm1.c:14] **BAD_PARM(incompatible)**
2        Wrong type passed to foo (argument 1: str)
         Expected char *, found int *.
>>                foo(iptr)
```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

Problem #2

Another simple problem occurs when arguments are passed to functions in the wrong order, as in the following example.

```
1:       /*
2:        * File: badparm2.c
3:        */
```

```
4:      long foo(f, l)
5:              double f;
6:              long l;
7:      {
8:              return f+l;
9:      }
10:
11:     main()
12:     {
13:             long ret = foo(32L, 32.0);
14:
15:             printf("%ld\n", ret);
16:             return 0;
17:     }
```

Diagnosis (during compilation)

```
1 [badparm2.c:13] **BAD_PARM(incompatible)**2
2       Wrong type passed to foo (argument 1: f)
        Expected double, found long.
>>              long ret = foo(32L, 32.0);

[badparm2.c:13] **BAD_PARM(incompatible)**
        Wrong type passed to foo (argument 2: l).
        Expected long, found double.
>>              long ret = foo(32L, 32.0);
```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

Problem #3

The following example illustrates the `BAD_PARM(union)` error category. The functions `func1` and `func2` expect to be passed a union and a pointer to an integer, respectively. The code in the `main` routine then invokes the two functions both properly and by passing the incorrect types.

Note that this code will probably work on most systems due to the internal alignment of the various data types. Relying on this behavior is, however, non-portable.

```
1:      /*
2:       * File: badparm4.c
3:       */
4:      union data {
5:              int i;
6:              double d;
7:      };
8:
9:      void func1(ptr)
10:             union data *ptr;
11:     {
12:             ptr->i = 1;
13:     }
14:
15:     void func2(p)
16:             int *p;
17:     {
18:             *p = 1;
19:     }
20:
21:     main()
22:     {
23:             int t;
24:             union data u;
25:
26:             func1(&u);
27:             func1(&t);          /* BAD_PARM */
28:             func2(&u);          /* BAD_PARM */
29:             func2(&t);
30:     }
```

Diagnosis (during compilation)

```
1 [badparm4.c:27] **BAD_PARM(union)**
2       Wrong type passed to func1 (argument 1: ptr)
        Expected union data *, found int *.
>>              func1(&t);                /* BAD_PARM */

[badparm4.c:28] **BAD_PARM(union)**
        Wrong type passed to func2 (argument 1: p)
        Expected int *, found union data *.
>>              func2(&u);                /* BAD_PARM */
```

1. Source lines at which problems were detected.
2. Description of the problem and the arguments that are incorrect.

Repair

Most of these problems are simple to correct based on the information given. For example, Problem #1 can be corrected by simply changing the code so that the pointer types (`char`, `int`) match. The other problems can be similarly corrected.

If your application generates error messages that you wish to ignore, you can suppress `BAD_PARM` in the Suppressions Control Panel.

This directive suppresses all `BAD_PARM` messages. If you wish to be more selective and suppress only a certain type of error, you can use the syntax

```
BAD_PARM(class1, class2, …)
```

where the arguments are one or more of the identifiers for the various categories of error described on “Mismatch In Argument Type” on page 219. Similarly, you can enable suppressed error messages by selecting **Unsuppress** in the **Action** field.

Thus, you could enable warnings about conflicts between types `int` and `long` (on systems where they are the same number of bytes) by unsuppressing

```
BAD_PARM(compatible)
```

In addition to the keywords described on “Mismatch In Argument Type” on page 219, you can also use the type `pointer` to suppress all messages about different pointer types.

For example, many programs declare functions with the argument type `char *`, which are then called with pointers to various other data types. The ANSI standard recommends that you use type `void *` in such circumstances, since this is allowed to match any pointer type. If, for some reason, you cannot do this, you can suppress messages from Insure++ about incompatible pointer types by suppressing

```
BAD_PARM(pointer)
```

in the Suppressions Control Panel.

COPY_BAD_RANGE

Copying Pointer Which Is Out-of-Range

This error is generated whenever an attempt is made to copy a pointer which points outside a valid range. It is not necessarily a serious problem, but may indicate faulty logic in the coding. Therefore, this error is suppressed by default.

Problem

The following code illustrates the problem in a simple way. In line 7, the pointer `a` is initialized as an array of 10 `char`s. The next line then attempts to make pointer `b` point to an area which has not been allocated. The resulting pointer is not valid.

```
1:       /*
2:        * File: copybad.cpp
3:        */
4:       int main() {
5:               char *a, *b;
6:
7:               a = new char [10];
8:               b = a + 20;
9:               return 0;
10:      }
```

Diagnosis (at runtime)

```
1 [copybad.cpp:8] **COPY_BAD_RANGE**
>>               b = a + 20;

2     Copying pointer which is out-of-range: a + 20

3     Pointer     : 0x0007c124
      Actual block: 0x0007c110 thru 0x0007c119 (10 bytes)
                 a, allocated at:
                          main()   copybad.cpp, 7
```

```
4    Stack trace where the error occurred:
                          main()   copybad.cpp, 8
```

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Description of the pointer which is out-of-range.
4. Stack trace showing the function call sequence leading to the error.

Repair

The simple way to avoid this problem is to not copy the invalid pointer. There may be an incorrect boundary case in your code causing the problem.

COPY_DANGLING

Copying pointer which has already been freed

This error is generated whenever an attempt is made to copy a pointer to a block of memory which has been freed. This error is suppressed by default.

Problem

The following code illustrates the problem in a simple way. In line 7, the pointer `a` is freed by calling `delete[]`. The next line then attempts to copy from the address `a` into the variable `b`. Since `a` has already been freed, `b` will not point to valid memory either.

```
1:        /*
2:         * File: copydang.cpp
3:         */
4:        int main() {
5:                char *a = new char [10], *b;
6:
7:                delete[] a;
8:                b = a;
9:                return 0;
10:       }
```

Diagnosis (at runtime)

```
1 [copydang.cpp:8] **COPY_DANGLING**
  >>              b = a;

2        Copying dangling pointer: a

3        Pointer  : 0x0007b6a0
         In block : 0x0007ebc0 thru 0x0007ebc9 (10 bytes)
                          a, allocated at:
                                  main()   copydang.cpp, 5

4                         stack trace where memory was
freed:
```

```
                                    main()   copydang.cpp, 7

5        Stack trace where the error occurred:
                                    main()   copydang.cpp, 8
```

1. Source line at which the problem was detected.
2. Brief description of the problem.
3. Description of the pointer which is dangling.
4. Stack trace showing where the dangling pointer was freed.
5. Stack trace showing the function call sequence leading to the error.

Repair

The simple way to avoid this problem is to not attempt to reuse pointers after they have been freed. Check that the deallocation that occurs at the indicated location should have taken place. Also check if pointer you are (mis)using should be pointing to a block allocated at the indicated place.

COPY_UNINIT_PTR

Copying Uninitialized Pointer

This error is generated whenever an uninitialized pointer is copied.

Note: This error category will be disabled if full uninitialized memory checking is in effect (the default). In this case, errors are detected in the `READ_UNINIT_MEM` category instead.

Problem

The pointer `a` is declared in line 5, but is never initialized. Therefore, when an attempt is made in line 7 to copy this pointer to `b`, an error is generated.

```
1:       /*
2:        * File: copyunin.cpp
3:        */
4:       int main() {
5:               char *a, *b;
6:
7:               b = a;
8:               return 0;
9:       }
```

Diagnosis (at runtime)

```
1 [copyunin.cpp:7] **COPY_UNINIT_PTR**
  >>             b = a;

2       Copying uninitialized pointer: a

3       Stack trace where the error occurred:
                         main()   copyunin.cpp, 7
```

1. Source line at which the problem was detected.
2. Brief description of the problem.

3. Stack trace showing the function call sequence leading to the error.

Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The example code given could be corrected by assigning a value to `a` before reaching line 7.

COPY_WILD

Copying Wild Pointer

This problem occurs when an attempt is made to copy a pointer whose value is invalid or which Insure++ did not see allocated.

This can come about in a couple of ways.

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

Note: This section focuses on the first type of problem described above. For information about the second type of problem, contact Parasoft's Quality Consultants.

Problem

The following code attempts to use the address of a variable but contains an error at line 9; the address operator (&) has been omitted.

```
1:	/*
2:	 * File: copywild.c
3:	 */
4:
5:	main()
6:	{
7:		int a = 123, *b;
8:
9:		b = a;
10:		return (0);
11:	}
```

Diagnosis (at runtime)

```
  [copywild.c:9] **COPY_WILD**
1 >>             b = a;

2       Copying wild pointer: a

3       Pointer : 0x0000007b
        Stack trace where the error occurred:
4                main() copywild.c, 9
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

DEAD_CODE

Code Is Not Executed

This error is generated when code is not evaluated, has no effect, or is unreachable. Insure++ distinguishes between several types of dead code as follows:

- `emptystmt` - The statement is empty.
- `emptyloopbody` - Loop body is empty.
- `noeffect` - Code has no effect.
- `notevaluated` - Code is not evaluated.

Error messages are classified according to this scheme and can be selectively enabled or disabled. By default, this error category is suppressed.

Problem #1

The following code shows a very tricky, well-disguised error that demonstrates how hard it is to find problems of this type without Insure++. The initialization function `get_glob` is never called by this code. Because `func_X2` is declared as a static function in the `Global` class, it can be called directly by `main`. This is in fact what happens, meaning that line 23 is interpreted as only a call to `func_X2`. Therefore, an error is generated since the call to `get_glob` is never evaluated.

```
1:      /*
2:       * File: deadcode.cpp
3:       */
4:      #include <iostream.h>
5:
6:      class Global {
7:              public:
8:              int j;
9:              static int func_X2(int i);
10:     };
11:
12:     int Global::func_X2(int i) {
13:             return i*2;
14:      }
15:
16:      Global *get_glob() {
17:               cerr << “Initializing...”
18:      << endl;
19:               return (Global *) 0;
20:      }
21:
22:      int main() {
23:               get_glob()->func_X2(2);
24:               return 0;
25:      }
```

Diagnosis (during compilation)

```
1 [deadcode.cpp:23] **DEAD_CODE(notevaluated)**
2        Code is not evaluated
  >>              get_glob()->func_X2(2);
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is incorrect.

Repair

This problem can be solved by replacing line 23 with a direct call to `func_X2`:

```
         Global::func_X2(2);
```

or by not making `func_X2` a static function.

In some cases, it may be that the dead code was never intended to be called. If that is the case, the dead code should be eliminated for clarity.

Problem #2

The following code illustrates several other types of DEAD_CODE errors, this time in C.

```
1:       /*
2:        * File: deadcode.c
3:        */
4:       int main()
```

```
5:      {
6:              int i = 0;
7:
8:              ;
9:              i;
10:             for (i; i; i)
11:                     ;
12:             return 0;
13:     }
```

Diagnosis (during compilation)

```
1 [deadcode.c:8] **DEAD_CODE(emptystmt)**
2       Statement is empty
  >>            ;
  [deadcode.c:9] **DEAD_CODE(noeffect)**
        Code has no effect
  >>            i;
  [deadcode.c:10] **DEAD_CODE(noeffect)**
        For loop initializer has no effect
  >>            for (i; i; i)
  [deadcode.c:10] **DEAD_CODE(noeffect)**
        For loop increment has no effect
  >>            for (i; i; i)
  [deadcode.c:11] **DEAD_CODE(emptyloopbody)**
        Loop body is empty (may be okay)
  >>            ;
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is incorrect.

Repair

These errors are usually corrected by removing the superfluous statement or by modifying the statement so that it does what it was intended to do, e.g., add a missing increment operator. An empty loop body may be useful in certain situations. In such a case, you may want to suppress that subcategory of `DEAD_CODE`.

DELETE_MISMATCH

Inconsistent Usage of Delete Operator

The current version of ANSI C++ distinguishes between memory allocated with `new` and `new[]`. A `delete` call *must* (according to the standard) match the `new` call, i.e. whether or not it has `[]`.

Calling `new[]` and `delete` may cause the compiler to not call the destructor on each element of the array, which can lead to serious errors. Even worse, if the memory was allocated differently, memory may be corrupted. This is definitely poor practice and unlikely to work with future releases of the specific compiler.

Problem #1

The following code shows a block of memory allocated with `new[]` and freed with `delete`, without `[]`.

```
1:       /*
2:        * File: delmis1.cpp
3:        */
4:
5:       int main() {
6:               int *a = new int [5];
7:               delete a;
8:               return 0;
9:       }
```

Diagnosis (at runtime)

```
1 [delmis1.cpp:7] **DELETE_MISMATCH**
  >>             delete a;

2        Inconsistent usage of delete operator: a

3        array deleted without []
                         a, allocated at:
                                 main()   delmis1.cpp, 6

4        Stack trace where the error occurred:
                                 main()   delmis1.cpp, 7
```

1. Source line at which the problem was detected.
2. Description of the problem and the operator which doesn't match.
3. Brief description of the mismatch.
4. Stack trace showing the function call sequence leading to the error.

Problem #2

The following code shows a block of memory allocated with `new`, without `[]`, and freed with `delete[]`. This may cause some implementations of C++ to crash, because the compiler may look for extra bits of information about how the block was allocated. Some compilers allow this type of error, extending the ANSI standard. In this case, there would be no extra bits, so the compiler would attempt to read from an invalid memory address.

```
1:        /*
2:         * File: delmis2.cpp
3:         */
4:
5:        int main() {
6:                int *a = new int;
7:                delete[] a;
8:                return 0;
9:        }
```

Diagnosis (at runtime)

```
1 [delmis2.cpp:7] **DELETE_MISMATCH**
  >>             delete[] a;

2       Inconsistent usage of delete operator: a

3       [] used to delete a non-array
                        a, allocated at:
                                main() delmis2.cpp, 6

4       Stack trace where the error occurred:
                                main()   delmis2.cpp, 7
```

1. Source line at which the problem was detected.
2. Description of the problem and the operator which doesn’t match.
3. Brief description of the mismatch.
4. Stack trace showing the function call sequence leading to the error.

Repair

To eliminate this error, you need to change the `delete` call to match the `new` call. In our first example, this could be accomplished by calling `delete[]` instead of `delete`, and vice versa in the second example.

EXPR_BAD_RANGE

Expression Exceeded Range

This error is generated whenever an expression uses a pointer that is outside its legal range. In many circumstances, these pointers are then turned into legal values before use (code generated by automated programming tools such as lex and yacc), so this error category is suppressed by default. If used with their illegal values, other Insure++ errors will be displayed which can be tracked to their source by re-enabling this error class.

Problem

In this code, the pointer a initially points to a character string. It is subsequently incremented beyond the end of the string. When the resulting pointer is used to make an array reference, a range error is generated.

```
1:        /*
2:         * File: exprange.c
3:         */
4:        main()
5:        {
6:                 char *a = "test";
7:                 char *b;
8:
9:                 a += 6;
10:                b = &a[1];
11:                return (0);
12:       }
```

Diagnosis (at runtime)

```
1 [exprange.c:10] **EXPR_BAD_RANGE**
  >>             b = &a[1];

2        Expression exceeded range: a[1]

         Index used: 1
```

```
3        Pointer           : 0x0000e226
         In block          : 0x0000e220 thru 0x0000e224 (5
bytes)
                             a, declared at exprange.c, 6

4        Stack trace where the error occurred:
                  main() exprange.c, 10
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the memory block to which the out of range pointer used to point, including the location at which it is declared.
4. Stack trace showing the function call sequence leading to the error.

Repair

In most cases, this error is caused by incorrect logic in the code immediately prior to that at which the message is generated. Probably the simplest method of solution is to run the program under a debugger with a breakpoint at the indicated location.

If you cannot find the error by examining the values of other variables at this location, the program should be run again, stopped somewhere shortly before the indicated line, and single-stepped until the problem occurs.

EXPR_DANGLING

Expression Uses Dangling Pointer

This error is generated whenever an expression operates on a dangling pointer - i.e., one which points to either

- A block of dynamically allocated memory that has already been freed.
- A block of memory which was allocated on the stack in some routine that has subsequently returned.

Problem

The following code fragment shows a block of memory being allocated and then freed. After the memory is de-allocated, the pointer to it is used again, even though it no longer points to valid memory.

```
1:       /*
2:        * File: expdangl.c
3:        */
4:       #include <stdlib.h>
5:
6:       main()
7:       {
8:                char *a = (char *)malloc(10);
9:                char b[10];
10:
11:               free(a);
12:               if(a > b)
13:                       a = b;
14:               return (0);
15:      }
```

Diagnosis (at runtime)

```
  [expdangl.c:12] **EXPR_DANGLING**
1 >>             if(a > b)

2     Expression uses dangling pointer: a > b
```

```
3     Pointer     : 0x00013868
      In block    : 0x00013868 thru 0x00013871 (10 bytes)
                          block allocated at:
                          malloc() (interface)
                                   main() expdangl.c, 8
                  stack trace where memory was freed:
                          main() expdangl.c, 11

4    Stack trace where the error occurred:
                          main() expdangl.c, 12
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the memory block to which the pointer used to point, including the location at which it was allocated and subsequently freed.
4. Stack trace showing the function call sequence leading to the error.

Repair

A good first check is to see if the pointer used in the expression at the indicated line is actually the one intended.

If it appears to be the correct pointer, check the line of code where the block was freed (as shown in the error message) to see if it was freed incorrectly.

EXPR_NULL

Expression Uses `NULL` Pointer

This error is generated whenever an expression operates on the `NULL` pointer.

Problem

The following code fragment declares a pointer, `a`, which is initialized to zero by virtue of being a global variable. It then manipulates this pointer, generating the `EXPR_NULL` error.

```
1:      /*
2:       * File: expnull.c
3:       */
4:      char *a;
5:
6:      main()
7:      {
8:              char *b;
9:
10:             b = &a[1];
11:             return (0);
12:     }
```

Diagnosis (at runtime)

```
1 [expnull.c:10] **EXPR_NULL**
  >>            b = &a[1];

2       Expression uses null pointer: a[1]
3       Stack trace where the error occurred:
                main() expnull.c, 10
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

Repair

One potential cause of this error is shown in this example. The pointer `a` is a global variable that will be initialized to zero by the compiler. Since this variable is never modified to point to anything else, it is still `NULL` when first used.

One way the given code can be corrected is by adding an assignment as follows

```
/*
 * File: expnull.c (modified)
 */
char *a;
main()
{
    char *b, c[10];
    a = c;
    b = &a[1];
    return (0);
}
```

It can also be corrected by allocating a block of memory.

A second possibility is that the pointer was set to zero by the program at some point before its subsequent use and not re-initialized. This is common in programs which make heavy use of dynamically allocated memory and which mark freed blocks by resetting their pointers to `NULL`.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the `RETURN_FAILURE` error code (see “RETURN_FAILURE” on page 322) and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

EXPR_UNINIT_PTR

Expression Uses Uninitialized Pointer

This error is generated whenever an expression operates on an uninitialized pointer.

Problem

The following code uses an uninitialized pointer.

```
/*
 * File: expuptr.c
 */
main()
{
        char *a, b[10], c[10];

        if (a > b)
                a = b;
        return (0);
}
```

Diagnosis (at runtime)

```
1 [expuptr.c:8] **EXPR_UNINIT_PTR**
  >>              if (a > b)

2        Expression uses uninitialized pointer: a > b
3        Stack trace where the error occurred:
                  main() expuptr.c, 8
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

Repair

This error is normally caused by omitting an assignment statement for the uninitialized variable. The example code can be corrected as follows:

```
/*
 * File: expuptr.c (modified)
 */
main()
{
        char *a, b[10], c[10];

        a = c;
        if (a > b)
                a = b;
        return (0);
}
```

EXPR_UNRELATED_ PTRCMP

Expression Compares Unrelated Pointers

This error is generated whenever an expression tries to compare pointers that do not point into the same memory block. This only applies to the operators >, >=, <, and <=. The operators == and ! = are exempt from this case.

The ANSI C-language specification declares this construct undefined except in the special case where a pointer points to an address one past the end of a block.

Problem

The following code illustrates the problem by comparing pointers to two data objects.

```
1:       /*
2:        * File: expucmp.c
3:        */
4:       #include <stdlib.h>
5:
6:       main()
7:       {
8:                char a[10], *b;
9:
10:               b = (char *)malloc(10);
11:
12:               if(a > b) a[0] = 'x';
13:               else a[0] = 'y';
14:               return (0);
15:      }
```

Note that the error in this code is not that the two objects `a` and `b` are of different data types (array vs. dynamic memory block), but that the comparison in line 12 attempts to compare pointers which do not point into the same memory block. According to the ANSI specification, this is an undefined operation.

Diagnosis (at runtime)

```
1 [expucmp.c:12] **EXPR_UNRELATED_PTRCMP**
  >>              if(a > b) a[0] = 'x';

2     Expression compares unrelated pointers: a > b

      Left hand side: 0xf7fffb8c
3     In block: 0xf7fffb8c thru 0xf7fffb95 (10 bytes)
                  a, declared at expucmp.c, 8

      Right hand side: 0x00013870
      In block: 0x00013870 thru 0x00013879 (10 bytes)
                  block allocated at:
                  malloc() (interface)
                    main() expucmp.c, 10
4     Stack trace where the error occurred:
                  main() expucmp.c, 12
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the two pointers involved in the comparison. For each pointer, the associated block of memory is shown together with its size and the line number at which it was declared or allocated.
4. Stack trace showing the function call sequence leading to the error.

Repair

While this construct is technically undefined according to the ANSI C specification, it is supported on many machines and its use is fairly common practice. If your application genuinely needs to use this construct, you can suppress this message by suppressing

```
        EXPR_UNRELATED_PTRCMP
```

in the Suppressions Control Panel.

EXPR_UNRELATED_ PTRDIFF

Expression Subtracts Unrelated Pointers

This error is generated whenever an expression tries to compute the difference between pointers that do not point into the same memory block.

The ANSI C language specification declares this construct undefined except in the special case where a pointer points to an object one past the end of a block.

Problem

The following code illustrates the problem by subtracting two pointers to different data objects.

```
1:      /*
2:       * File: expudiff.c
3:       */
4:      #include <stdlib.h>
5:
6:      main()
7:      {
8:              char a[10], *b;
9:              int d;
10:
11:             b = (char *)malloc(10);
12:             d = b - a;
13:             return (0);
14:     }
```

Diagnosis (at runtime)

```
  [expudiff.c:12] **EXPR_UNRELATED_PTRDIFF**
1 >>            d = b - a;

2       Expression subtracts unrelated pointers: b - a
```

```
        Left hand side    : 0x00013878
3       In block : 0x00013878 thru 0x00013881 (10 bytes)
                        b, allocated at:
                        malloc()  (interface)
                          main()  expudiff.c, 11

        Right hand side   : 0xf7fffb8c
        In block : 0xf7fffb8c thru 0xf7fffb95 (10 bytes)
                        a, declared at expudiff.c, 8
4       Stack trace where the error occurred:
                main() expudiff.c, 12
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the two pointers involved in the expression. For each pointer the associated block of memory is shown together with its size and the line number at which it was declared or allocated.
4. Stack trace showing the function call sequence leading to the error.

Repair

While this construct is undefined according to the ANSI C language specification, it is supported on many machines and its use is fairly common practice. If your application genuinely needs to use this construct, you can suppress error messages by suppressing

```
EXPR_UNRELATED_PTRDIFF
```

in the Suppressions Control Panel.

EXPR_WILD

Expression Uses Wild Pointer

This error is generated whenever a program operates on a memory region that is unknown to Insure++. This can come about in two ways:

- Errors in user code that result in pointers that don’t point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

Note: This section focuses on the first type of problem described here.For information about the second type of problem, contact ParaSoft’s Quality Consultants.

Problem #1

The following code attempts to use the address of a local variable but contains an error at line 8 - the address operator (&) has been omitted.

```
1:       /*
2:        * File: expwld1.c
3:        */
4:       main()
5:       {
6:                int i = 123, j=345, *a;
7:
8:                a = i;
9:                if(a > &i)
10:                        a = &j;
11:               return (0);
12:      }
```

Diagnosis (at runtime)

```
  [expwld1.c:9] **EXPR_WILD**
1 >>              if(a > &i)

2        Expression uses wild pointer: a > &i
```

```
3        Pointer : 0x0000007b

4        Stack trace where the error occurred:
                          main() expwld1.c,
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the wild pointer.
4. Stack trace showing the function call sequence leading to the error.

Keep in mind that most compilers will generate warning messages for this error since the assignment in line 8 uses incompatible types.

Problem #2

A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally attempting to use it.

```
/*
 * File: expwld2.c
 */
union {
        int *ptr;
        int ival;
} u;

main()
{
        int i = 123, j=345;

        u.ptr = &i;
        u.ival = i;
        if(u.ptr > &j)
                u.ptr = &j;
        return (0);
18:      }
```

Note that this code will not generate compile time errors.

Diagnosis (at runtime)

```
  [expwld2.c:15] **EXPR_WILD**
1 >>              if(u.ptr > &j)

2        Expression uses wild pointer: u.ptr > &j

3        Pointer : 0x0000007b

         Stack trace where the error occurred:
4                         main() expwld2.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Repair

The simpler types of problem are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step it through the code leading up to the error.

“Wild pointers” can also be generated when Insure++ has only partial information about your program’s structure. For more information on this topic, contact ParaSoft’s Quality Consultants.

FREE_BODY

Freeing Memory Block From Body

This error is generated when an attempt is made to de-allocate memory by using a pointer which currently points into the middle of a block, rather than to its beginning.

Problem

The following code attempts to free a memory region using an invalid pointer.

```
/*
 * File: freebody.c
 */
#include <stdlib.h>

main()
{
         char *a = (char *)malloc(10);
         free(a+1);
}
```

Diagnosis (at runtime)

```
  [freebody.c:9] **FREE_BODY**
1 >>             free(a+1);

2        Freeing memory block from body: a + 1

3        Pointer      : 0x000173e9
         Stack trace where the error occurred:
4                     main() freebody.c, 9

5        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the pointer that is being deallocated.

4. Stack trace showing the function call sequence leading to the error.
5. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

This is normally a serious error. In most cases, the line number indicated in the diagnostics will have a simple error that can be corrected.

FREE_DANGLING

Freeing Dangling Pointer

This error is generated when a memory block is freed multiple times.

Problem

The following code frees the same pointer twice.

```
/*
 * File: freedngl.c
 */
#include <stdlib.h>

main()
{
         char *a = (char *)malloc(10);
         free(a);
         free(a);
         return (0);
}
```

Diagnosis (at runtime)

```
  [freedngl.c:10] **FREE_DANGLING**
1 >>             free(a);

2        Freeing dangling pointer: a

3        Pointer  : 0x000173e0
         In block : 0x000173e0 thru 0x000173e9 (10 bytes)
                          block allocated at:
4                                  malloc() (interface)
                                     main() freedngl.c, 8

5                 stack trace where memory was freed:
                          main() freedngl.c, 9

6        Stack trace where the error occurred:
                 main() freedngl.c, 10
```

```
7        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the pointer that is being deallocated.
4. Information about the block of memory addressed by this pointer, including information about where this block was allocated.
5. Stack trace showing where this block was freed.
6. Stack trace showing the function call sequence leading to the error.
7. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems allow memory blocks to be freed multiple times. However, this is not portable and is not a recommended practice.

The information supplied in the diagnostics will allow you to see the line of code which previously de-allocated this block of memory. You should attempt to remove one of the two calls.

If your application is unable to prevent multiple calls to deallocate the same block, you can suppress error messages by suppressing

```
FREE_DANGLING
```

in the Suppressions Control Panel.

FREE_GLOBAL

Freeing Global Memory

This error is generated if the address of a global variable is passed to a routine that de-allocates memory.

Problem

The following code attempts to deallocate a global variable that was not dynamically allocated.

```
1:       /*
2:        * File: freeglob.c
3:        */
4:       char a[10];
5:
6:       main()
7:       {
8:                free(a);
9:                return (0);
10:      }
```

Diagnosis (at runtime)

```
  [freeglob.c:8] **FREE_GLOBAL**
1 >>              free(a);

2        Freeing global memory: a

3        Pointer  : 0x00012210
         In block : 0x00012210 thru 0x00012217 (8 bytes)
4                           a,declared at freeglob.c, 4

         Stack trace where the error occurred:
5                          main() freeglob.c, 8

6        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the pointer that is being deallocated.
4. Information about the block of memory addressed by this pointer, including information about where this block was declared.
5. Stack trace showing the function call sequence leading to the error.
6. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems allow this operation, since they keep track of which blocks of memory are actually dynamically allocated, but this is not portable programming practice and is not recommended.

In some cases, this error will result from a simple coding mistake at the indicated source line which can be quickly corrected.

A more complex problem may arise when a program uses both statically and dynamically allocated blocks in the same way. A common example is a linked list in which the head of the list is static, while the other entries are allocated dynamically. In this case, you must take care not to free the static list head when removing entries.

If your application is unable to distinguish between global and dynamically allocated memory blocks, you can suppress error messages by suppressing

```
        FREE_GLOBAL
```

in the Suppressions Control Panel.

FREE_LOCAL

Freeing Local Memory

This error is generated if the address of a local variable is passed to `free`.

Problem

The following code attempts to free a local variable that was not dynamically allocated.

```
1:        /*
2:         * File: freelocl.c
3:         */
4:        main()
5:        {
6:                char b, *a;
7:
8:                a = &b;
9:                free(a);
10:               return (0);
11:       }
```

Diagnosis (at runtime)

```
  [freelocl.c:9] **FREE_LOCAL**
1 >>              free(a);

2        Freeing local memory: a

3        Pointer           : 0xf7fffb0f
         In block          : 0xf7fffb0f thru 0xf7fffb0f (1
byte)
4                            b,declared at freelocl.c, 6
         Stack trace where the error occurred:
5                 main() freelocl.c, 9

6        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the pointer that is being deallocated.
4. Information about the block of memory addressed by this pointer, including information about where this block was declared.
5. Stack trace showing the function call sequence leading to the error.
6. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems allow this operation since they keep track of which blocks of memory are actually dynamically allocated, but this is not portable programming practice and is not recommended.

In most cases, this error will result from a simple coding mistake at the indicated source line which can be quickly corrected.

If your application is unable to distinguish between local variables and dynamically allocated memory blocks, you can suppress error messages by suppressing

```
        FREE_LOCAL
```

in the Suppressions Control Panel.

FREE_UNINIT_PTR

Freeing Uninitialized Pointer

This error is generated whenever an attempt is made to de-allocate memory by means of an uninitialized pointer.

Problem

This code attempts to free a pointer which has not been initialized.

```
1:        /*
2:         * File: freeuptr.c
3:         */
4:        main()
5         {
6:                 char *a;
7:                 free(a);
8:                 return (0);
9:        }
```

Diagnosis (at runtime)

```
  [freeuptr.c:7] **FREE_UNINIT_PTR**
1 >>               free(a);

2         Freeing uninitialized pointer: a
          Stack trace where the error occurred:
3                  main() freeuptr.c, 7

4         **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.
4. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems appear to allow this operation, since they will refuse to free memory that was not dynamically allocated. Relying on this behavior is very dangerous, however, since an uninitialized pointer may “accidentally” point to a block of memory that *was* dynamically allocated, but should not be freed.

FREE_WILD

Freeing Wild Pointer

This error is generated when memory is de-allocated that is unknown to Insure++. This can come about in two ways:

- Errors in user code that result in pointers that don’t point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

Note: This section focuses on the first type of problem described here. For information on the second type of problem, contact ParaSoft’s Quality Consultants.

A particularly unpleasant problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally attempting to free the initial memory block.

```
/*
 * File: freewild.c
 */
#include <stdlib.h>

union {
         int *ptr;
         int ival;
} u;

main()
{
         char *a = (char *)malloc(100);

         u.ptr = a;
         u.ival = 123;
         free(u.ptr);
         return (0);
}
```

Diagnosis (at runtime)

```
  [freewild.c:17] **FREE_WILD**
1 >>              free(u.ptr);

2        Freeing wild pointer: u.ptr

3        Pointer : 0x0000007b

         Stack trace where error occurred:
4                         main() freewild.c, 17
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Repair

This problem is most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step through the code leading up to the problem.

“Wild pointers” can also be generated when Insure++ has only partial information about your program’s structure. Contact ParaSoft’s Quality Consultants for more information on this topic.

FUNC_BAD

Function Pointer Is Not a Function

This error is generated when an attempt is made to call a function through either an invalid or unknown function pointer.

Problem

One simple way to generate this error is through the use of the `union` data type. If the union contains a function pointer which is invoked after initializing some other union member, this error can occur.

```
1:   /*
2:    * File: funcbad.c
3:    */
4:   union {
5:        int *iptr;
6:        int (*fptr)();
7:   } u;
8:
9:   main()
10:  {
11:       int i;
12:
13:       u.iptr = &i;
14:       u.fptr();
15:       return (0);
16:  }
```

Diagnosis (at runtime)

```
[funcbad.c:14] **FUNC_BAD**
1 >>              u.fptr();

2         Function pointer is not a function: u.fptr

3         Pointer           : 0xf7fff8cc
          In block          : 0xf7fff8cc thru 0xf7fff8cf
4                                    (4 bytes,1 element)
```

```
                                    i, declared at func-
bad.c, 11
         Stack trace where the error occurred:
5                  main() funcbad.c, 14

6                  **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. The value of the pointer through which the call is being attempted.
4. Description of the memory block to which this pointer actually points, including its size and the source line of its declaration.
5. Stack trace showing the function call sequence leading to the error.
6. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

The description of the memory block to which the pointer points should enable you to identify the statement which was used to assign the function pointer incorrectly.

FUNC_NULL

Function Pointer Is `NULL`

This error is generated when a function call is made via a `NULL` function pointer.

Problem

This code attempts to call a function through a pointer that has never been explicitly initialized. Since the pointer is a global variable, it is initialized to zero by default, resulting in the attempt to call a `NULL` pointer.

```
/*
 * File: funcnull.c
 */
void (*a)();

main()
{
         a();
         return (0);
}
```

Diagnosis (at runtime)

```
  [funcnull.c:8] **FUNC_NULL**
1 >>              a();

2        Function pointer is null: a
         Stack trace where the error occurred:
3                 main() funcnull.c, 8

4        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

3. Stack trace showing the function call sequence leading to the error.
4. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

The most common way to generate this problem is the one shown here, in which the pointer never was explicitly initialized and is set to zero. This case normally requires the addition of an assignment statement prior to the call as shown below.

```
/*
 * File: funcnull.c (modified)
 */
void (*a)();
extern void myfunc();

main()
{
    a = myfunc;
    a();
    return (0);
}
```

A second fairly common programming practice is to terminate arrays of function pointers with `NULL` entries. Code that scans a list looking for a particular function may end up calling the `NULL` pointer if its search criterion fails. This normally indicates that protective programming logic should be added to prevent against this case.

FUNC_UNINIT_PTR

Function Pointer Is Uninitialized

This error is generated when a call is made through an uninitialized function pointer.

Problem

This code attempts to call a function through a pointer that has not been set.

```
1:       /*
2:        * File: funcuptr.c
3:        */
4:       main()
5:       {
6:                void (*a)();
7:
8:                a();
9:                return (0);
10:      }
```

Diagnosis (at runtime)

```
  [funcuptr.c:8] **FUNC_UNINIT_PTR**
1 >>            a();

2        Function pointer is uninitialized: a
         Stack trace where the error occurred:
3                 main() funcuptr.c, 8

4        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

4. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

This problem normally occurs because some assignment statement has been omitted from the code. The current example can be fixed as follows:

```
extern void myfunc();

main()
{
    void (*a)();
    a = myfunc;
    a();
}
```

INSURE_ERROR

Internal Errors (Various)

This error code is reserved for fatal errors that Insure++ is unable to deal with adequately, such as running out of memory or failing to open a required file.

Unrecognized string values in the Windows Registry or Advanced Options can also generate this error.

INSURE_WARNING

Errors From `iic_warning` Calls

This error code is generated when Insure++ encounters a call to the `iic_warning` interface function.

Example

The following code contains a call to a function called `archaic_function` whose use is to be discouraged.

```
/*
 * File: warn.c
 */
#include <stdio.h>

main()
{
        archaic_function();
        exit(0);
}
```

In order to use the `iic_warning` capability, we can make an interface to the `archaic_function` as follows.

```
/*
 * File: warn_i.c
 */
void archaic_function(void)
{
  iic_warning(
        "This function is obsolete");
  archaic_function();
}
```

Diagnosis (during compilation)

```
1 [warn.c:8] **INSURE_WARNING**
2       Use of archaic_function is deprecated.
  >>              archaic_function();
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

Repair

This error category is suppressed by default, so you must unsuppress

```
        INSURE_WARNING
```

in the Suppressions Control Panel before compiling code which uses it.

There are many uses for `iic_warning` and the `INSURE_WARNING` error, so no specific suggestions for error correction are appropriate. Hopefully, the messages displayed by the system will provide sufficient assistance.

LEAK_ASSIGN

Memory Leaked Due To Pointer Reassignment

This error is generated whenever a pointer assignment occurs which will prevent a block of dynamically allocated memory from ever being freed. Normally this happens because the pointer being changed is the only one that still points to the dynamically allocated block.

Problem

This code allocates a block of memory, but then reassigns the pointer to the block to a static memory block. As a result, the dynamically allocated block can no longer be freed.

```
1:       /*
2:        * File: leakasgn.c
3:        */
4:       #include <stdlib.h>
5:
6:       main()
7:       {
8:                char *b, a[10];
9:
10:               b = (char *)malloc(10);
11:               b = a;
12:               return (0);
13:      }
```

Diagnosis (at runtime)

```
  [leakasgn.c:11] **LEAK_ASSIGN**
1 >>            b = a;

2       Memory leaked due to pointer reassignment: <return>

3       Lost block:      0x000173e8 thru 0x000173f1 (10
bytes)
                                       block allocated at:
                                       malloc() (interface)
```

```
                                      main() leakasgn.c, 10

         Stack trace where the error occurred:
4                 main() leakasgn.c, 11
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
4. Stack trace showing the function call sequence leading to the error.

Repair

In many cases, this problem is caused by simply forgetting to free a previously allocated block of memory when a pointer is reassigned. For example, the leak in the example code can be corrected as follows:

```
10:      b = (char *)malloc(10);
11:      free(b);
12:      b = a;
```

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages, suppress

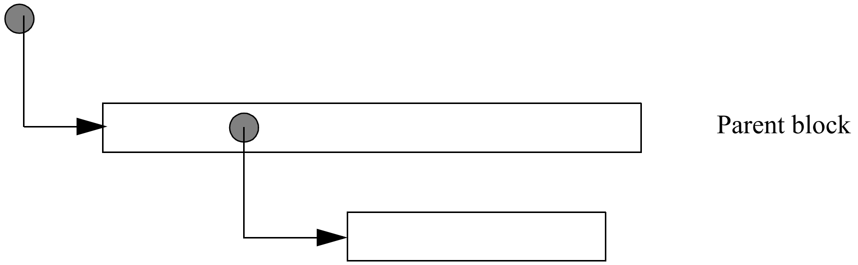
```
LEAK_ASSIGN
```

in the Suppressions Control Panel.

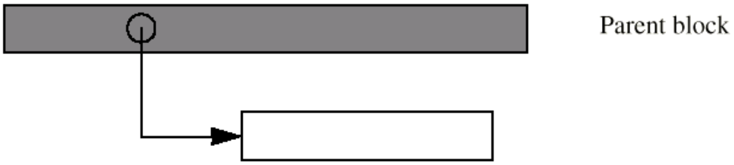
LEAK_FREE

Memory Leaked Freeing Block

This problem can occur when a block of memory contains a pointer to another dynamically allocated block, as indicated in the following figure.



If the main memory block is freed its memory becomes invalid, which means that the included pointer can no longer be used to free the second block. This causes a permanent memory leak.



Problem

This code defines PB to be a data structure that contains a pointer to another block of memory.

```
1:      /*
2:       * File: leakfree.c
3:       */
4:      #include <stdlib.h>
5:
6:      typedef struct ptrblock {
7:              char *ptr;
8:      } PB;
9:
10:     main()
11:     {
12:             PB *p;
13:
14:             p = (PB *)malloc(sizeof(*p));
15:             p->ptr = malloc(10);
16:
17:             free(p);
18:             return (0);
19:     }
```

We first create a single `PB` and then allocate a block of memory for it to point to. The call to `free` on the `PB` then causes a permanent memory leak, since it frees the memory containing the only pointer to the second allocated block. This latter block can no longer be freed.

Diagnosis (at runtime)

```
  [leakfree.c:17] **LEAK_FREE**
1 >>            free(p);

2       Memory leaked freeing block: <return>

3       Lost block:       0x00013888 thru 0x00013891 (10
bytes)
                          block allocated at:
                          malloc() (interface)
                            main() leakfree.c, 15
```

```
          Stack trace where the error occurred:
4                   main() leakfree.c, 17
```

1. Source line at which the problem was detected.
2. Description of the problem and the value that is about to be lost.
3. Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
4. Stack trace showing the function call sequence leading to the error.

Repair

In many cases, this problem is caused by forgetting to free the enclosed blocks when freeing their container. This can be corrected by adding a suitable call to free the memory before freeing the parent block.

Caution must be used when doing this, however, to ensure that the memory blocks are freed in the correct order. Changing the example in the following manner, for example, would still generate the same error:

```
free(p);
free(p->ptr);
```

because the blocks are freed in the wrong order. The contained blocks must be freed before their parents, because the memory becomes invalid as soon as it is freed. Thus, the second call to `free` in the above code fragment might fail, because the value `p->ptr` is no longer valid. It is quite legal, for example, for the first call to `free` to have set to zero or otherwise destroyed the contents of its memory block. (Many systems allow the out of order behavior, although it is becoming less portable as more and more systems move to dynamically re-allocated (moveable) memory blocks.)

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case suppress

```
LEAK_FREE
```

in the Suppressions Control Panel.

LEAK_RETURN

Memory Leaked By Ignoring Returned Value

This error is generated whenever a function returns a pointer to a block of memory which is then ignored by the calling routine. In this case, the allocated memory block is permanently lost and can never be freed.

Problem

This code calls the function `gimme`, which returns a memory block that is subsequently ignored by the `main` routine.

```
1:       /*
2:        * File: leakret.c
3:        */
4:       #include <stdlib.h>
5:
6:       char *gimme()
7:       {
8:                return malloc(10);
9:       }
10:
11:      main()
12:      {
13:               gimme();
14:               return (0);
15:      }
```

Diagnosis (at runtime)

```
  [leakret.c:8] **LEAK_RETURN**
1 >>            gimme();

2        Memory leaked ignoring return value: <return>

3        Lost block:       0x000173e8 thru 0x000173f1 (10
bytes)
                           block allocated at:
                           malloc()  (interface)
```

```
                   gimme()  leakret.c, 8
                    main()  leakret.c, 13

Stack trace where the error occurred:
         main() leakret.c, 13
```

1. Source line at which the problem was detected.
2. Description of the problem and the block that is to be lost.
3. Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.

Repair

This problem usually results from an oversight on the part of the programmer, or a misunderstanding of the nature of the pointer returned by a routine. In particular, it is sometimes unclear whether the value returned points to a static block of memory, which will not need to be freed, or a dynamically allocated one, which should be.

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case, suppress

```
LEAK_RETURN
```

in the Suppressions Control Panel.

LEAK_SCOPE

Memory Leaked Leaving Scope

This error is generated whenever a function allocates memory for its own use and then returns without freeing it or saving a pointer to the block in an external variable. The allocated block can never be freed.

Problem

This code calls the function `gimme`, which allocates a memory block that is never freed.

```
1:       /*
2:        * File: leakscop.c
3:        */
4:       #include <stdlib.h>
5:
6:       void gimme()
7:       {
8:               char *p;
9:               p = malloc(10);
10:              return;
11:      }
12:
13:      main()
14:      {
15:              gimme();
16:              return (0);
17:      }
```

Diagnosis (at runtime)

```
  [leakscop.c:10] **LEAK_SCOPE**
1 >>             return;

2        Memory leaked leaving scope: <return>

3        Lost block:      0x0003870 thru 0x00013879 (10
bytes)
```

```
                          block allocated at:
                          malloc()  (interface)
                           gimme()  leakscop.c, 9
                            main()  leakscop.c, 15
         Stack trace where the error occurred:
4                 gimme()   leakscop.c, 10
                  main() leakscop.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the block that is to be lost.
3. Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
4. Stack trace showing the function call sequence leading to the error.

Repair

This problem usually results from an oversight on the part of the programmer and is cured by simply freeing a block before returning from a routine. In the current example, a call to `free(p)` before line 10 would cure the problem.

A particularly easy way to generate this error is to return from the middle of a routine, possibly due to an error condition arising, without freeing previously allocated data. This bug is easy to introduce when modifying existing code.

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case, suppress

```
LEAK_SCOPE
```

in the Suppressions Control Panel.

PARM_BAD_RANGE

Array Parameter Exceeded Range

This error is generated whenever a function parameter is declared as an array, but has more elements than the actual argument which was passed.

Problem

The following code fragment shows an array declared with one size in the `main` routine and then used with another in a function.

```
/*
 * File: parmrnge.c
 */
int foo(a)
        int a[10];
{
        return a[5];
}

int b[5];

main()
{
        int a;
        a = foo(b);
        return (0);
}
```

Diagnosis (at runtime)

```
  [parmrnge.c:6] **PARM_BAD_RANGE**
1 >> {

2        Array parameter exceeded range: a

3                          bbbbbb
                           | 20 | 20 |
                           pppppppppp
```

```
4        Parameter (p)      0xf7fffb04 thru 0xf7fffb2b (40
bytes)
         Actual block (b)           0xf7fffb04 thru
0xf7fffb17
                                    (20 bytes, 5 elements)
5                          b, declared at parmrnge.c, 10

         Stack trace where the error occurred:
                  foo() parmrnge.c, 6
6                 main() parmrnge.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Schematic showing the relative layout of the memory block which was actually passed as the argument (b) and expected parameter (p). (See “Overflow Diagrams” on page 191.)
4. Description of the memory range occupied by the parameter, including its length.
5. Description of the actual block of data corresponding to the argument, including its address range and size. Also includes the name of the real variable which matches the argument and the line number at which it was declared.
6. Stack trace showing the function call sequence leading to the error.

Repair

This error is normally easy to correct based on the information presented in the diagnostic output.

The simplest solution is to change the definition of the array in the called routine to indicate an array of unknown size, i.e., replace line 5 with

```
parmrnge.c, 5            int a[];
```

This declaration will match any array argument and is the recommended approach whenever the called routine will accept arrays of variable size.

An alternative is to change the declaration of the array in the calling routine to match that expected. In this case, line 10 could be changed to

```
parmrnge.c, 10           int b[10];
```

which now matches the argument declaration.

PARM_DANGLING

Array Parameter Is Dangling Pointer

This error is generated whenever a parameter declared as an array is actually passed a pointer to a block of memory that has been freed.

Problem

The following code frees its memory block before passing it to `foo`.

```
1:        /*
2:         * File: parmdngl.c
3:         */
4:        #include <stdlib.h>
5:
6:        char foo(a)
7:                 char a[10];
8:        {
9:                 return a[0];
10:       }
11:
12:       main()
13:       {
14:                char *a;
15:                a = malloc(10);
16:                free(a);
17:                foo(a);
18:                return (0);
19:       }
```

Diagnosis (at runtime)

```
  [parmdngl.c:8] **PARM_DANGLING**
1 >>      {

2         Array parameter is dangling pointer: a

3         Pointer  : 0x0001adb0
4         In block : 0x0001adb0 thru 0x0001adb9 (10 bytes)
                   block allocated at:
                          malloc() (interface)
                            main() parmdngl.c, 15

                  stack trace where memory was freed:
5                           main() parmdngl.c, 16

         Stack trace where the error occurred:
                             foo() parmdngl.c, 8
6                           main() freedngl.c,17
```

1. Source line at which the problem was detected.
2. Description of the problem and the parameter that is in error.
3. Value of the pointer that was passed and has been deallocated.
4. Information about the block of memory addressed by this pointer, including information about where this block was allocated.
5. Indication of the line at which this block was freed.
6. Stack trace showing the function call sequence leading to the error.

Repair

This error is normally caused by freeing a piece of memory too soon. A good strategy is to examine the line of code indicated by the diagnostic message which shows where the memory block was freed and check that it should indeed have been de-allocated.

A second check is to verify that the correct parameter was passed to the subroutine.

A third strategy which is sometimes useful is to `NULL` pointers that have been freed and then check in the called subroutine for this case. Code similar to the following is often useful

```
#include <stdlib.h>

char foo(a)
         char *a;
{
         if(a) return a[0];
         return '!';
}

main()
{
         char *a;
         a = (char *)malloc(10);
         free(a);
         a = NULL;
         foo(a);
         return (0);
}
```

The combination of resetting the pointer to `NULL` after freeing it and the check in the called subroutine prevents misuse of dangling pointers.

PARM_NULL

Array Parameter Is NULL

This error is generated whenever a parameter declared as an array is actually passed a NULL pointer.

Problem

The following code fragment shows a function which is declared as having an array parameter, but which is invoked with a NULL pointer. The value of array is NULL because it is a global variable, initialized to zero by default.

```
1:       /*
2:        * File: parmnull.c
3:        */
4:       int foo(a)
5:               int a[];
6:       {
7:               return 12;
8:       }
9:
10:      int *array;
11:
12:      main()
13:      {
14:              foo(array);
15:              return (0);
16:      }
```

Diagnosis (at runtime)

```
[parmnull:6] **PARM_NULL**
1 >> {

2       Array parameter is null: a
        Stack trace where the error occurred:
3               foo() parmnull.c, 6
                main() parmnull.c, 14
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Stack trace showing the function call sequence leading to the error.

Repair

A common cause of this error is the one given in this example, a global pointer which is initialized to zero by the compiler and then never reassigned. The correction for this case is to include code to initialize the pointer, possibly by allocating dynamic memory or by assigning it to some other array object.

For example, we could change the `main` routine of the example to

```
main()
{
    int local[10];

    array = local;
    foo(array);
}
```

This problem can also occur when a pointer is set to `NULL` by the code (perhaps to indicate a freed block of memory) and then passed to a routine that expects an array as an argument.

In this case, Insure++ distinguishes between functions whose arguments are declared as arrays

```
int foo(int a[])
{
```

and those with pointer arguments

```
int foo(int *a)
{
```

The latter type will not generate an error if passed a `NULL` argument, while the former will.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the `RETURN_FAILURE` error code (see “RETURN_FAILURE” on page 322) and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

If your application cannot avoid passing a `NULL` pointer to a routine, you should either change the declaration of its argument to the second style or suppress these error messages by suppressing

```
PARM_NULL
```

in the Suppressions Control Panel.

PARM_UNINIT_PTR

Array Parameter Is Uninitialized Pointer

This error is generated whenever an uninitialized pointer is passed as an argument to a function which expects an array parameter.

Problem

This code passes the uninitialized pointer `a` to routine `foo`.

```
1:       /*
2:        * File: parmuptr.c
3:        */
4:       char foo(a)
5:                char a[10];
6:       {
7:                return a[0];
8:       }
9:
10:      main()
11:      {
12:               char *a;
13:
14:               foo(a);
15:               return (0);
16:      }
```

Diagnosis (at runtime)

```
  [parmuptr.c:6] **PARM_UNINIT_PTR**
1 >>     {

2        Array parameter is uninitialized pointer: a

         Stack trace where the error occurred:
3                 foo()    parmuptr.c, 6
                  main() parmuptr.c, 14
```

1. Source line at which the problem was detected.

2. Description of the problem and the argument that is in error.
3. Stack trace showing the function call sequence leading to the error

Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given, for example, could be corrected by including an assignment as shown below.

```
/*
 * File: parmuptr.c (Modified)
 */
...
main()
{
    char *a, b[10];
    a = b;
    foo(a);
}
```

PARM_WILD

Array Parameter Is Wild

This error is generated whenever a parameter is declared as an array but the actual value passed when the function is called points to no known memory block.

This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

Note: This section focuses on the first type of problem described above. For information about the second type of problem, contact ParaSoft's Quality Consultants.

Problem #1

The following code attempts to pass the address of a local variable to the routine `foo` but contains an error at line 14 - the address operator (`&`) has been omitted.

```
1:        /*
2:         * File: parmwld1.c
3:         */
4:        void foo(a)
5:                 int a[];
6:        {
7:                 return;
8:        }
9:
10:       main()
11:       {
12:                int i = 123, *a;
13:
14:                a = i;
15:                foo(a);
16:               return (0);
17:       }
```

Diagnosis (at runtime)

```
  [parmwld1.c:6] **PARM_WILD**
1 >> {

2       Array parameter is wild: a

3       Pointer : 0x0000007b

                Stack trace where the error occurred:
4                                foo() parmwld1.c, 6
                                main() parmwld1.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

Problem #2

A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally passing it to a function.

```
1:      /*
2:       * File: parmwld2.c
3:       */
4:      union {
5:              int *ptr;
6:              int ival;
7:      } u;
```

```
8:
9:      void foo(a)
10:             int a[];
11:     {
12:             return;
13:     }
14:
15:     main()
16:     {
17:             int i = 123;
18:
19:             u.ptr = (int *)&i;
20:             u.ival = i;
21:             foo(u.ptr);
22:             return (0);
23:     }
```

Note that this code will not generate compile time errors.

Diagnosis (at runtime)

```
  [parmwld2.c:11] **PARM_WILD**
1 >> {

2       Array parameter is wild: a

3       Pointer : 0x0000007b

                Stack trace where the error occurred:
4                                 foo() parmwld2.c, 11
                                 main() parmwld2.c, 21
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Repair

This problem is most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step through the code leading up to the problem.

Note that wild pointers can also be generated when Insure++ has only partial information about your program’s structure. For more information about this topic, contact ParaSoft’s Quality Consultants.

READ_BAD_INDEX

Reading Array Out-of-Range

This error is generated whenever an illegal value will be used to index an array. It is a particularly common error that can be very difficult to detect, especially if the out-of-range elements happen to have zero values.

If this error can be detected during compilation, an error will be issued instead of the normal runtime error.

Problem

This code attempts to access an illegal array element due to an incorrect loop range.

```
1:       /*
2:        * File: readindx.c
3:        */
4:       int a[10];
5:       int junk;
6:       main()
7:       {
8:               int i, tot=0;
9:
10:              for(i=1; i<=10; i++)
11:                      tot += a[i];
12:              return (0);
13:      }
```

Diagnosis (at runtime)

```
  [readindx.c:11] **READ_BAD_INDEX**
1 >>             tot += a[i];

2        Reading array out of range: a[i]

3        Index used: 10

4        Valid range: 0 thru 9 (inclusive)
```

```
          Stack trace where the error occurred:
5                   main() readindx.c, 11
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Illegal index value used.
4. Valid index range for this array.
5. Stack trace showing the function call sequence leading to the error.

Repair

Typical sources of this error include loops with incorrect initial or terminal conditions, as in this example, for which the corrected code is:

```
main()
{
    int i, tot=0, a[10];

    for(i=0; i<sizeof(a)/sizeof(a[0]); i++)
        tot += a[i];
    return (0);
}
```

READ_DANGLING

Reading From a Dangling Pointer

This problem occurs when an attempt is made to dereference a pointer that points to a block of memory that has been freed.

Problem

This code attempts to use a piece of dynamically allocated memory after it has already been freed.

```
1:   /*
2:    * File: readdngl.c
3:    */
4:   #include <stdlib.h>
5:
6:   main()
7:   {
8:        char b;
9:        char *a = (char *)malloc(10);
10:
11:       free(a);
12:       b = *a;
13:       return (0);
14:  }
```

Diagnosis (at runtime)

```
  [readdngl.c:12] **READ_DANGLING**
1 >>              b = *a;

2        Reading from a dangling pointer: a

3        Pointer: 0x000173e8
4        In block:0x000173e8 thru 0x000173f1 (10 bytes)
                  block allocated at:
                           malloc() (interface)
                             main() readdngl.c, 9
```

```
5                 stack trace where memory was freed:
                             main() readdngl.c, 11

         Stack trace where the error occurred:
6                 main() readdngl.c, 12
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the dangling pointer variable.
4. Description of the block to which this pointer used to point, including its size, name and the line at which it was allocated.
5. Stack trace showing where this block was freed.
6. Stack trace showing the function call sequence leading to the error.

Repair

Check that the de-allocation that occurs at the indicated location should, indeed, have taken place. Also check that the pointer you are using should really be pointing to a block allocated at the indicated place.

READ_NULL

Reading NULL pointer

This error is generated whenever an attempt is made to dereference a NULL pointer.

Problem

This code attempts to use a pointer which has not been explicitly initialized. Since the variable a is global, it is initialized to zero by default, which results in dereferencing a NULL pointer in line 10.

```
1:       /*
2:        * File: readnull.c
3:        */
4:       int *a;
5:
6:       main()
7:       {
8:               int b, c;
9:
10:              b = *a;
11:      }
```

Diagnosis (at runtime)

```
  [readnull.c:10] **READ_NULL**
1 >>             b = *a;

2        Reading null pointer: a
         Stack trace where the error occurred:
3                main() readnull.c, 10

4        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

3. Stack trace showing the function call sequence leading to the error.
4. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

A common cause of this problem is the one shown in the example - use of a pointer that has not been assigned and which is initialized to zero. This is usually due to the omission of an assignment or allocation statement which would give the pointer a reasonable value.

The example code might, for example, be corrected as follows:

```
/*
 * File: readnull.c (modified)
 */
int *a;

main()
{
        int b, c;

        a = &c;
        b = *a;
}
```

A second common source of this error is code which dynamically allocates memory, but then zeroes pointers as blocks are freed. In this case, the error would indicate reuse of a freed block.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the `RETURN_FAILURE` error code (see “RETURN_FAILURE” on page 322) through the Suppressions Control Panel and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

READ_OVERFLOW

Reading Overflows Memory

This error is generated whenever a read operation would access a piece of memory beyond the valid range for a block.

Problem #1

This code attempts to copy a string into the array `b`. Note that although the array is large enough, the `memcpy` operation will fail, since it attempts to read past the end of the string `a`.

```
1:   /*
2:    * File: readovr1.c
3:    */
4:   main()
5:   {
6:        char *a = "TEST";
7:        char b[20];
8:
9:        memcpy(b, a, sizeof(b));
10:       return (0);
11:  }
```

Diagnosis (at runtime)

```
[readovr1.c:9] **READ_OVERFLOW**

1>>               memcpy(b, a, sizeof(b));

2        Reading overflows memory: <argument 2>

                  bbbbb
3                 | 5 |    15    |
                  rrrrrrrrrrrrrrrr

         Reading (r): 0x00012218 thru 0x0001222b (20 bytes)
```

```
4      From block(b):  0x00012218 thru 0x0001221c (5 bytes)
                  a, declared at readovr1.c, 6

5        Stack trace where the error occurred:
                  memcpy() (interface)
                  main() readovr1.c, 9
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Schematic showing the relative layout of the actual memory block (`b`) and region being read (`r`) (see “Overflow Diagrams” on page 191).
4. Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.
5. Stack trace showing the function call sequence leading to the error.

Problem #2

A second fairly common case arises when strings are not terminated properly. The code shown below copies a string using the `strncpy` routine, which leaves it non-terminated since the buffer is too short. When we attempt to print this message, an error results.

```
/*
 * File: readovr2.c
 */
main()
{
         char junk;
         char b[8];
         strncpy(b, "This is a test",
                            sizeof(b));
         printf("%s\n", b);
         return (0);
}
```

Diagnosis (at runtime)

```
  [readovr2.c:10] **READ_OVERFLOW**
1 >>              printf("%s\n", b);

2        String is not null terminated within range: b

3        Reading           : 0xf7fffb50
4        From block: 0xf7fffb50 thru 0xf7fffb57 (8 bytes)
                          b, declared at readovr2.c, 7

         Stack trace where the error occurred:
5                 main() readovr2.c, 10
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Pointer being used as a string.
4. Block from which the read is taking place, including its size and the location of its declaration.
5. Stack trace showing the function call sequence leading to the error.

A slight variation on this misuse of strings occurs when the pointer, passed as a string, lies completely outside the range of its buffer. In this case, the diagnostics will appear as above except that the description line will contain the message

```
Alleged string does not begin within legal range
```

Problem #3

This code attempts to read past the end of the allocated memory block by reading the second element of the union.

```
/*
 * File: readovr3.c
 */
#include <stdlib.h>

struct small {
    int x;
8:        };
9:
10:       struct big {
11:           double y;
12:       };
13:
14:       union two
15:       {
16:           struct small a;
17:           struct big b;
18:       };
19:
20:       int main()
21:       {
22:           struct small *var1;
23:           union two *ptr;
24:           double d;
25:
26:           var1 = (struct small *)malloc (sizeof(struct
small));
27:           ptr = (union two *) var1;
28:           d = ptr->b.y;
29:           return (0);
30:       }
```

Diagnosis (at runtime)

```
  [readovr3.c:28] **READ_OVERFLOW**
1 >>              d = ptr->b.y;

2         Structure reference out of range: ptr

                  bbbbb
3                 | 4 | 4 |
                  rrrrrrrrr

     Reading (r):   0x0001fce0 thru 0x0001fce7 (8 bytes)
4    From block(b):  0x0001fce0 thru 0x0001fce3 (4 bytes)
                           block allocated at:
                                    malloc() (interface)
                                    main() readovr3.c, 26
```

```
          Stack trace where the error occurred:
5                   main() readovr3.c, 28
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Schematic showing the relative layout of the actual memory block (`b`) and region being read (`r`). (See “Overflow Diagrams” on page 191.)
4. Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.
5. Stack trace showing the function call sequence leading to the error.

Problem #4

This code shows a C++ problem that can occur when using inheritance and casting pointers incorrectly.

```
/*
 * File: readover.cpp
 */
#include <stdlib.h>

class small
{
public:
         int x;
};

class big : public small
{
public:
         double y;
};

int main()
{
```

```
20:               small *var1;
21:               big *var2;
22:               double d;
23:
24:               var1 = new small;
25:               var2 = (big *) var1;
26:               d = var2->y;
27:               return (0);
28:      }
```

Diagnosis (at runtime)

```
  [readover.cpp:26] **READ_OVERFLOW**
1 >>              d = var2->y;

2        Structure reference out of range: var2

                  bbbbb
3                 | 4 | 4 |  8  |
                          rrrrrrr

         Reading  (r): 0x0001fce0 thru 0x0001fce7 (8 bytes)
4        From block(b): 0x0001fce0 thru 0x0001fce3 (4 bytes)
                                  var1, allocated at:
                         operator new()
                                  main() readover.cpp, 24
         Stack trace where the error occurred:
5                 main() readover.cpp, 26
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Schematic showing the relative layout of the actual memory block (b) and region being read (r). (See “Overflow Diagrams” on page 191.)
4. Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.

5. Stack trace showing the function call sequence leading to the error.

Repair

These errors often occur when reading past the end of a string or using the `sizeof` operator incorrectly. In most cases, the indicated source line contains a simple error.

The code for problem #1 could, for example, be corrected by changing line 9 to

```
        memcpy(b, a, strlen(a)+1);
```

READ_UNINIT_MEM

Reading Uninitialized Memory

The use of uninitialized memory is a difficult problem to isolate, since the effects of the problem may not show up until much later. This problem is complicated by the fact that quite a lot of references to uninitialized memory are harmless.

To deal with these issues, Insure++ distinguishes two sub-categories of the `READ_UNINIT_MEM` error class

- `copy` - This error code is generated whenever an application assigns a variable using an uninitialized value. In itself, this may not be a problem, since the value may be reassigned to a valid value before use or may never be used. This error category is suppressed by default.
- `read` - This code is generated whenever an uninitialized value is used in an expression or some other context where it must be incorrect. This error category is enabled by default, but is detected only if the `checking_uninit` option is `on`.
  (see “Advanced Configuration Options Used by Insure++” on page 167)

The difference between these two categories is illustrated in the following examples.

Note: Full checking may be disabled by setting the option `checking_uninit off` (see “Advanced Configuration Options Used by Insure++” on page 167). If full uninitialized memory checking is disabled, uninitialized pointers will still be detected, but will be reported in the `READ_UNINIT_PTR` category (see “READ_UNINIT_PTR” on page 315).

Problem #1

This code attempts to use a structure element which has never been initialized.

```
/*
 * File: readuni1.c
 */
```

```
4:       #include <stdio.h>
5:
6:       main()
7:       {
8:               struct rectangle {
9:                       int width;
10:                      int height;
11:              };
12:
13:              struct rectangle box;
14:              int area;
15:
16:              box.width = 5;
17:              area = box.width*box.height;
18:              printf("area = %d\n", area);
19:              return (0);
20:      }
```

Diagnosis (at runtime)

```
  [readuni1.c:17] **READ_UNINIT_MEM(read)**
1 >>             area = box.width * box.height;

2        Reading uninitialized memory: box.height
         Stack trace where the error occurred:
3                main() readuni1.c, 17
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

Problem #2

This code assigns the value `b` using memory returned by the `malloc` system call, which is uninitialized.

```
1:       /*
2:        * File: readuni2.c
3:        */
4:       #include <stdlib.h>
```

```
5:
6:        main()
7:        {
8:                 int *a = (int *)malloc(5);
9:                 int b;
10:
11:                b = *a;
12:                return (0);
13:       }
```

The code in line 11 of this example falls into the `copy` error sub-category, since the uninitialized value is merely used to assign another variable. If **b** were later used in an expression, it would then generate a `READ_UNINIT_MEM(read)` error.

Note: If the `int`s in lines 8 and 9 of the above example were replaced by `char`s, the error would not be detected by default. To see the error in the new example, you would need to set the option `checking_uninit_min_size 1`. For more information about this option, see “Advanced Configuration Options Used by Insure++” on page 167.

Diagnosis (at runtime)

```
  [readuni2.c:11] **READ_UNINIT_MEM(copy)**
1 >>              b = *a;

         Reading uninitialized memory: *a

         In block: 0x00062058 thru 0x0006205c (5 bytes)
                  block allocated at:
                          malloc()  (interface)
                            main()  readuni2.c, 8

         Stack trace where the error occurred:
3                 main() readuni2.c, 11
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

Repair

As mentioned earlier, the `READ_UNINIT_MEM(copy)` error category is suppressed by default, so you will normally only see errors in the `read` category. In many cases, these will be errors that can be simply corrected by initializing the appropriate variables. In other cases, these values will have been assigned from other uninitialized variables, which can be detected by unsuppressing the `copy` sub-category and running again.

READ_UNINIT_PTR

Reading From Uninitialized Pointer

This error is generated whenever an uninitialized pointer is dereferenced.

Note: This error category will be disabled if full uninitialized memory checking is in effect (the default). In this case, errors are detected in the `READ_UNINIT_MEM` category instead. (see “READ_UNINIT_MEM” on page 311)

Problem

This code attempts to use the value of the pointer a, even though it has never been initialized.

```
/*
 * File: readuptr.c
 */
main()
{
        int b, *a;

        b = *a;
        return (0);
}
```

Diagnosis (at runtime)

```
  [readuptr.c:8] **READ_UNINIT_PTR**
1 >>            b = *a;

2       Reading from uninitialized pointer: a

        Stack trace where the error occurred:
3                       main() readuptr.c, 8
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given can be corrected by including an assignment as shown below.

```
/*
 * File: readuptr.c (Modified)
 */
main()
{
    int b, *a, c;

    a = &c;
    b = *a;
    return (0);
}
```

READ_WILD

Reading Wild Pointer

This problem occurs when an attempt is made to dereference a pointer whose value is invalid or which Insure++ did not see allocated.

This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

Note: This section focuses on the first type of problem described here. For information on the second type of problem, contact ParaSoft's Quality Consultants.

Problem #1

The following code attempts to use the address of a variable but contains an error at line 8 - the address operator (`&`) has been omitted.

```
1:      /*
2:       * File: readwld1.c
3:       */
4:      main()
5:      {
6:              int *a, i = 123, b;
7:
8:              a = i;
9:              b = *a;
10:             return (0);
11:     }
```

Diagnosis (at runtime)

```
[readwld1.c:9] **READ_WILD**
1>>             b = *a;
```

```
2       Reading wild pointer: a

3       Pointer : 0x0000007b

        Stack trace where the error occurred:
4                main() readwld1.c, 9
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

Problem #2

A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before using it.

```
/*
 * File: readwld2.c
 */
union {
         int *ptr;
         int ival;
} u;

main()
{
         int b, i = 123;

         u.ptr = &i;
         u.ival = i;
         b = *u.ptr;
         return (0);
}
```

Note that this code will not generate compile time errors.

Diagnosis (at runtime)

```
  [readwld2.c:15] **READ_WILD**
1 >>              b = *u.ptr;

2        Reading wild pointer: u.ptr

3        Pointer : 0x0000007b

         Stace trace where error occurred:
4                          main() readwld2.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Repair

The simpler types of problem are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point shortly before the error and single-step through the code leading up to the problem.

Note that wild pointers can also be generated when Insure++ has only partial information about your program’s structure. For more information on this topic, contact ParaSoft’s Quality Consultants.

RETURN_DANGLING

Returning Pointer To Local Variable

This error is generated whenever a function returns a pointer to a (non-static) local variable. Since the stack frame of this routine will disappear when the function returns, this pointer is never valid.

Problem

The following code shows the routine `foo` returning a pointer to a local variable.

```
1:      /*
2:       * File: retdngl.c
3:       */
4:      char *foo()
5:      {
6:              char b[10];
7:              return b;
8:      }
9:
10:     main()
11:     {
12:             char *a = foo();
13:             return 0;
14:     }
```

Diagnosis (during compilation)

```
1 [retdngl.c:7] **RETURN_DANGLING**
2       Returning pointer to local variable: b.
>>              return b;
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

Repair

The pointer returned in this manner can be made legal in one of several ways.

- Passing the required buffer from the calling function, and if required also passing the size of the buffer as yet another parameter
- Declaring the memory block `static` in the called routine, i.e., line 6 would become

  ```
  static char b[10];
  ```

- Allocating a block dynamically instead of on the stack and returning a pointer to it, e.g.,

  ```
  char *foo()
  {
      return malloc(10);
  }
  ```

- Making the memory block into a global variable rather than a local one.

Occasionally, the value returned from the function is never used in which case it is safest to change the declaration of the routine to indicate that no value is returned.

RETURN_FAILURE

Function Call Returned An Error

A particularly difficult problem to track with conventional methods is that of incorrect return code from system functions. Insure++ is equipped with interface definitions for system libraries that enable it to check for errors when functions are called. Normally, these messages are suppressed, since applications often include their own handling for system calls that return errors. In some cases, however, it may be useful to enable these messages to track down totally unexpected behavior.

Problem

A particularly common problem occurs when applications run out of memory as in the following code.

```
/*
 * File: retfail.c
 */
#include <stdlib.h>

main()
{
        char *p;

        p = malloc(1024*1024*1024);
        return (0);
}
```

Diagnosis

Normally, this code will run without displaying any messages. If RETURN_FAILURE messages are enabled, however, the following display will result.

```
  [retfail.c:10] **RETURN_FAILURE**
1 >>              p = malloc(1024*1024*1024);

        Function returned an error:
2                 malloc(1073741824) failed: no more memory

        Stack trace where the error occurred:
3                 malloc() (interface)
                  main() retfail.c, 10
```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.
3. Stack trace showing the function call sequence leading to the error.

Repair

These messages are normally suppressed, but can be enabled by unsuppressing

```
        RETURN_FAILURE
```

in the Suppressions Control Panel.

RETURN_INCONSISTENT

Function Has Inconsistent Return Type

Insure++ checks that each function returns a result consistent with its declared data type, and that a function with a declared return type actually returns an appropriate value.

Because there are several different ways in which functions and return values can be declared, Insure++ divides up this error category into four levels or subcategories as follows:

- Level 1 - Function has no explicitly declared return type (and so defaults to `int`) and returns no value. (This error level is normally suppressed.)
- Level 2 - Function is explicitly declared to return type `int` but returns nothing.
- Level 3 - Function explicitly declared to return a data type other than `int` but returns no value.
- Level 4 - The function returns the value for one data type at one statement and another data type at another statement.

In many applications, errors at levels 1 and 2 need to be suppressed, since older codes often include these constructs.

Problem

The following code demonstrates the four different error levels.

```
/*
 * File: retinc.c
 */
func1() {
        return;
}

int func2() {
        return;
}

```

```
12:      double func3() {
13:               return;
14:      }
15:
16:      int func4(a)
17:               int a;
18:      {
19:               if (a < 3) return a;
20:               return;
21:      }
```

Diagnosis (During compilation)

```
1 [retinc.c:4] **RETURN_INCONSISTENT(1)**
2        Function func1 has an inconsistent return type.
         Declared return type implicitly "int",
                  but returns no value.
>> func1() {
[retinc.c:8] **RETURN_INCONSISTENT(2)**
         Function func2 has an inconsistent return type.
         Declared return type "int", but returns no value.
>> int func2() {
[retinc.c:12] **RETURN_INCONSISTENT(3)**
         Function func2 has an inconsistent return type.
         Declared return type "double", but returns no
value.
>> double func3() {
[retinc.c:20] **RETURN_INCONSISTENT(4)**
         Function func4 has an inconsistent return type.
         Returns value in one location, and not in another.
>> return;
```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.

Repair

As already suggested, older codes often generate errors at levels 1 and 2 which are not particularly serious. You can either correct these problems by adding suitable declarations or suppress them by suppressing

```
        RETURN_INCONSISTENT(1, 2)
```

in the Suppressions Control Panel.

Errors at levels 3 and 4 should probably be investigated and corrected.

UNUSED_VAR

Unused Variables

Insure++ has the ability to detect unused variables in your code. Since these are not normally errors, but informative messages, this category is disabled by default.

Two different sub-categories are distinguished.

- `assigned` - The variable is assigned a value but never used.
- `unused` - The variable is never used.

Problem #1

The following code assigns a value to the variable `max` but never uses it.

```
1:       /*
2:        * File: unuasign.c
3:        */
4:       main()
5:       {
6:               int i, a[10];
7:               int max;
8:
9:               a[0] = 1;
10:              a[1] = 1;
11:              for(i=2; i<10; i++)
12:                      a[i] = a[i-1]+a[i-2];
13:              max = a[9];
14:      }
```

Diagnosis (during compilation)

Normally this code will run without displaying any messages. If `UNUSED_VAR` messages are enabled, however, the following display will result.

```
1 [unuasign.c:7] **UNUSED_VAR(assigned)**
2       Variable assigned but never used: max
>               int max;
```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.

Problem #2

The following code never uses the variable `max`.

```
1:       /*
2:        * File: unuvar.c
3:        */
4:       main()
5:       {
6:                int i, a[10];
7:                int max;
8:
9:                a[0] = 1;
10:               a[1] = 1;
11:               for(i=2; i<10; i++)
12:                        a[i] = a[i-1]+a[i-2];
13:      }
```

Diagnosis (during compilation)

If `UNUSED_VAR` messages are enabled, however, the following display will result.

```
1 [unuvar.c:7] **UNUSED_VAR(unused)**
2        Variable declared but never used: max
>>                int max;
```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.

Repair

These messages are normally suppressed but can be enabled by unsuppressing

```
UNUSED_VAR
```

in the Suppressions Control Panel.

You can also enable each sub-category independently by unsuppressing

```
UNUSED_VAR(assigned)
```

In most cases, the corrective action to be taken is to remove the offending statement, since it is not affecting the behavior of the application. In certain circumstances, these errors might denote logical program errors in which a variable should have been used but wasn't.

USER_ERROR

User Generated Error Message

This error is generated when a program violates a rule specified in an interface module. These normally check that parameters passed to system level or user functions fall within legal ranges or are otherwise valid. This behavior is different from the `RETURN_FAILURE` error code, which normally indicates that the call to the function was made with valid data, but that it still returned an error for some, possibly anticipated, reason.

Problem

These problems fall into many different categories. A particularly simple example is shown in the following code, which calls the `sqrt` function and passes it a negative argument.

```
/*
 * File: usererr.c
 */
#include <math.h>

main()
{
         double q;

         q = sqrt(-2.0);
         return (0);
}
```

Diagnosis (at runtime)

```
  [usererr.c:10] **USER_ERROR**
1 >>              q = sqrt(-2.0);

2        Negative number -2.000000 passed to sqrt:

         Stack trace where the error occurred:
3                          main() usererr.c, 10
```

1. Source line at which the problem was detected.
2. Description of the error and the parameters used.
3. Stack trace showing the function call sequence leading to the error.

Repair

Each message in this category is caused by a different problem, which should be evident from the printed diagnostic. Usually, these checks revolve around the legality of various arguments to functions.

These messages can be suppressed by suppressing

```
USER_ERROR
```

in the Suppressions Control Panel

VIRTUAL_BAD

Error In Runtime Initialization Of Virtual Functions

This error is caused when a virtual function has not been initialized prior to being used by another function.

Problem

The following pieces of code illustrate this error. The virtual function `func` is declared in `virtbad1.cpp` in the `goo` class. A static variable of this class, `barney`, is also declared in that file. The function crash calls `func` through `barney` in line 23. In file `virtbad2.cpp`, a static variable of class `foo`, `fred`, is declared. Class `foo` calls `crash`, which then in turn ends up calling the virtual function `func`. A virtual function's address is not established until the program is initialized at runtime, and static functions are also initialized at runtime. This means that depending on the order of initialization, `fred` could be trying to find `func`, which does not yet have an address. The `VIRTUAL_BAD` error message is generated when this code is compiled with Insure++.

Note: Due to differences in the object layout of different compilers, this error might not be detected with certain compilers.

```
1:        /*
2:         * File: virtbad1.cpp
3:         */
4:        #include <iostream>
5:
6:        class goo {
7:                public:
8:                int i;
9:                goo::goo() {
10:                       cerr << "goo is initialized."
                                  << endl; }
11:               virtual int func();
12:               virtual int func2();
13:       };
14:       static goo barney;
```

```
int crash() {
        int ret;
        cerr << “Sizeof(goo) = ” <<
                           sizeof(goo) << endl;
        cerr << “Sizeof(i) = “ <<
                           sizeof(int) << endl;
        char *cptr = (char *) &barney;
        cptr += 4;
        long *lptr = (long *) cptr;
        cerr << “vp = “ << *lptr << endl;
        ret = barney.func();
        cerr << “crash” << endl;
        return ret;
}
int goo::func() {
        cerr << “goo.func” << endl;
        func2();
        return i;
}
int goo::func2() {
        cerr << “goo.func2” << endl;
        return 2;
}
```

Figure 1. virtbad1.C

```
/*
 * File: virtbad2.cpp
 */
#include <iostream>

extern int crash();

class foo {
        public:
        foo::foo() {
                cerr << “foo” << endl;
                cerr << “Got “ <<
                           crash() << endl;
        }
};

```

```
16:      static foo fred;
```

Figure 2. virtbad2.C

```
1:       /*
2:        * File: virtbad3.cpp
3:        */
4:       #include <iostream>
5:
6:       int main() {
7:                cerr << “main” << endl;
8:                return 0;
9:       }
```

Figure 3. virtbad3.C

Diagnosis (at runtime)

```
[virtbad1.cpp:29] **VIRTUAL_BAD**
1 >>             func2();

2        Virtual function table is invalid: func2()

3        Stack trace where the error occurred:
                 goo::func()virtbad1.cpp, 29
                 crash()  virtbad1.cpp, 23
                 foo::foo()virtbad2.cpp, 12
         __mod_I__fred0virtbad21001_cc_000()
                 _main()
                 main()   virtbad3.cpp, 6

  **Memory corrupted. Program may crash!!**
4 Abort (core dumped)
```

1. Source line at which the problem was detected.
2. Description of the problem and which virtual function caused the error.
3. Stack trace showing the function call sequence leading to the error.
4. Core dumps typically follow these messages, as any usage of the dynamic memory functions will be unable to cope.

Repair

The error in the sample code could be eliminated by not making `fred` static. In that case, the address for `func` would be generated during the initialization before any requests for it existed, and then no problems would occur.

WRITE_BAD_INDEX

Writing Array Out-of-Range

This error is generated whenever an illegal value will be used to index an array which is being written.

If this error can be detected during compilation, a compilation error will be issued instead of the normal runtime error.

Problem

This code attempts to access an illegal array element due to an incorrect loop range.

```
1:       /*
2:        * File: writindx.c
3:        */
4:       main()
5:       {
6:                int i, a[10];
7:
8:                for(i=1; i<=10; i++)
9:                         a[i] = 0;
10:               return (0);
11:      }
```

Diagnosis (at runtime)

```
  [writindx.c:9] **WRITE_BAD_INDEX**
1 >>             a[i] = 0;

2        Writing array out of range: a[i]

3        Index used: 10

4        Valid range: 0 thru 9 (inclusive)
         Stack trace where the error occurred:
5                          main() writindx.c, 9

6        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Illegal index value used.
4. Valid index range for this array.
5. Stack trace showing the function call sequence leading to the error.
6. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

This is normally a fatal error and is often introduced algorithmically.

Other typical sources include loops with incorrect initial or terminal conditions, as in this example, for which the corrected code is:

```
main()
{
    int i, a[10];

    for(i=; i<sizeof(a)/sizeof(a[0]); i++)
        a[i] = 0;
    return (0);
}
```

WRITE_DANGLING

Writing To a Dangling Pointer

This problem occurs when an attempt is made to dereference a pointer that points to a block of memory that has been freed.

Problem

This code attempts to use a piece of dynamically allocated memory after it has already been freed.

```
1:       /*
2:        * File: writdngl.c
3:        */
4:       #include <stdlib.h>
5:
6:       main()
7:       {
8:                char *a = (char *)malloc(10);
9:
10:               free(a);
11:               *a = 'x';
12:               return (0);
13:      }
```

Diagnosis (at runtime)

```
  [writdngl.c:11] **WRITE_DANGLING**
1 >>              *a = 'x';

2        Writing to a dangling pointer: a

3        Pointer: 0x000173e8
4        In block:0x000173e8 thru 0x000173f1 (10 bytes)
                  block allocated at:
                           malloc() (interface)
                             main() writdngl.c, 8
5                 stack trace where memory was freed:
                             main()  writdngl.c, 10
```

```
6        Stack trace where the error occurred:
                  main() writdngl.c, 11

         **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Value of the dangling pointer variable
4. Description of the block to which this pointer used to point, including its size, name, and the line at which it was allocated.
5. Indication of the line at which this block was freed.
6. Stack trace showing the function call sequence leading to the error.

Repair

Check that the de-allocation that occurs at the indicated location should indeed have taken place. Also check that the pointer you are using should really be pointing to a block allocated at the indicated place.

WRITE_NULL

Writing To a NULL Pointer

This error is generated whenever an attempt is made to dereference a NULL pointer.

Problem

This code attempts to use a pointer which has not been explicitly assigned. Since the variable a is global, it is initialized to zero by default, which results in dereferencing a NULL pointer in line 8.

```
1:       /*
2:        * File: writnull.c
3:        */
4:       int *a;
5:
6:       main()
7:       {
8:                *a = 123;
9:                return (0);
10:      }
```

Diagnosis (at runtime)

```
  [writnull.c:8] **WRITE_NULL**
1 >>             *a = 123;

2       Writing to a null pointer: a
        Stack trace where the error occurred:
3                         main() writnull.c, 8

4       **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.

3. Stack trace showing the function call sequence leading to the error.
4. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

A common cause of this problem is the one shown in the example-- use of a pointer that has not been explicitly assigned and which is initialized to zero. This is usually due to the omission of an assignment or allocation statement which would give the pointer a reasonable value.

The example code might, for example, be corrected as follows

```
/*
 * File: writnull.c (Modified)
 */
int *a;

main()
{
         int b;

         a = &b;
         *a = 123;
         return (0);
}
```

A second common source of this error is code which dynamically allocates memory but then zeroes pointers as blocks are freed. In this case, the error would indicate reuse of a freed block.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the `RETURN_FAILURE` error code (see “RETURN_FAILURE” on page 322) via your Suppressions Control Panel and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

WRITE_OVERFLOW

Writing Overflows Memory

This error is generated whenever a block of memory indicated by a pointer will be written outside its valid range.

Problem

This code attempts to copy a string into the array `a`, which is not large enough.

```
1:       /*
2:        * File: writover.c
3:        */
4:       main()
5:       {
6:                int junk;
7:                char a[10];
8:
9:                strcpy(a, "A simple test");
10:               return (0);
11:      }
```

Diagnosis (at runtime)

```
  [writover.c:9] **WRITE_OVERFLOW**
1 >>            strcpy(a, "A simple test");

2        Writing overflows memory: a

                bbbbbbbbbb
3               |   10   |  4  |
                wwwwwwwwwwwwwwww

4        Writing (w): 0xf7fffafc thru 0xf7fffb09 (14 bytes)
         To block (b):      0xf7fffafc thru 0xf7fffb05 (10
bytes)

                             a, declared at writover.c, 7
```

```
5       Stack trace where the error occurred:
                  strcpy () (interface)
                  main() writover.c, 9
```

1. Source line at which the problem was detected.
2. Description of the problem and the incorrect expression.
3. Schematic showing the relative layout of the actual memory block (`b`) and region being written (`w`). (See “Overflow Diagrams” on page 191.)
4. Range of memory being written and description of the block to which the write is taking place, including its size and the location of its declaration.
5. Stack trace showing the call sequence leading to the error.

Repair

This error often occurs when working with strings. In most cases, a simple fix is to increase the size of the destination object.

WRITE_UNINIT_PTR

Writing To An Uninitialized Pointer

This error is generated whenever an uninitialized pointer is dereferenced.

Problem

This code attempts to use the value of the pointer a, even though it has not been initialized.

```
1:       /*
2:        * File: writuptr.c
3:        */
4:       main()
5:       {
6:                int *a;
7:
8:                *a = 123;
9:                return (0);
10:      }
```

Diagnosis (at runtime)

```
  [writuptr.c:8] **WRITE_UNINIT_PTR**
1 >>              *a = 123;

2        Writing to an uninitialized pointer: a

         Stack trace where the error occurred:
3                          main() writuptr.c, 8

4        **Memory corrupted. Program may crash!!**
```

1. Source line at which the problem was detected.
2. Description of the problem and the expression that is in error.
3. Stack trace showing the function call sequence leading to the error.

4. Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given, for example, could be corrected by including an assignment as shown below.

```
         /*
          * File: writuptr.c (Modified)
          */
         main()
         {
             int *a, b;

             a = &b;
             *a = 123;
}
```

WRITE_WILD

Writing To a Wild Pointer

This problem occurs when an attempt is made to dereference a pointer whose value is invalid or which Insure++ did not see allocated.

This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

Note: This section focuses on the first type of problem described here. For information about the second type of problem, contact ParaSoft's Quality Consultants.

Problem #1

The following code attempts to use the address of a variable but contains an error at line 8 - the address operator (&) has been omitted.

```
1:      /*
2:       * File: writwld1.c
3:       */
4:      main()
5:      {
6:              int i = 123, *a;
7:
8:              a = i;
9:              *a = 99;
10:             return (0);
11:     }
```

Diagnosis (at runtime)

```
  [writwld1.c:9] **WRITE_WILD**
1 >>              *a = 99;
```

```
2        Writing to a wild pointer: a

3        Pointer : 0x0000007b

4        Stack trace where the error occurred:
                           main() writwld1.c, 9
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment in line 8 uses incompatible types.

Problem #2

A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before using it.

```
/*
 * File: writwld2.c
 */
union {
         int *ptr;
         int ival;
} u;

main()
{
         int i = 123;

         u.ptr = &i;
         u.ival = i;
         *u.ptr = 99;
         return (0);
}
```

Note that this code will not generate compile time errors.

Diagnosis (at runtime)

```
[writwld2.c:15] **WRITE_WILD**
1>>               *u.ptr = 99;

2        Writing to a wild pointer: u.ptr

3        Pointer : 0x0000007b

4        Stack trace where the error occurred:
                            main() writwld2.c, 15
```

1. Source line at which the problem was detected.
2. Description of the problem and the name of the parameter that is in error.
3. Value of the bad pointer.
4. Stack trace showing the function call sequence leading to the error.

Repair

The simpler types of problems are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively, you can stop the program at some point shortly before the error and single-step through the code leading up to the problem.

Note that wild pointers can also be generated when Insure++ has only partial information about your program’s structure. For more information about this topic, contact ParaSoft’s Quality Consultants.

Index

Symbols

Numerics

A